CS205 Object Oriented Programming in Java

Module 5 - **Graphical User Interface and Database support of Java**

(Part 1)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Swings**
  - ☑ Swings fundamentals
  - ☑ Swing Key Features

Swing fundamentals

- Swing is **written entirely in Java** (platform-independent).
  - So swing components are **light-weight**
  - Swing components are NOT implemented by platform-specific code.
- **Swing** is a **set of classes**.
- Swing is built on the foundation of the AWT(Abstract Window Toolkit).
- Swing provides more **powerful and flexible** functionalities than standard AWT components.
- Swing classes are defined in javax.swing package and its subpackages.

Swing Key Features

- Two key features of Swing are
  - Swing components are **Lightweight**
  - Swing supports a **Pluggable Look and Feel**

Swing Key Features-**Swing components are Lightweight**

- Swing Components are **lightweight** because
  - they are **written entirely in Java**
  - they do **not** map directly to **platform-specific peers.**
    - Peer classes are written by java API developers to interface with native objects
  - Lightweight components do not call the native operating system for drawing the graphical user interface(GUI) components
  - They are **rendered using graphics primitives**
    - they can be transparent, which enables nonrectangular shapes.
  - lightweight components are more efficient and more flexible.

Swing Key Features-**Swing components are Lightweight**(contd.)

- Lightweight components
  - **do not translate into native peers**,
  - the **look and feel of each component is determined by Swing**, not by the underlying operating system.
- This means that **each component will work in a consistent manner across all platforms.**

Swing Key Features-**Swing Supports a Pluggable Look and Feel**

- Swing supports a **pluggable look and feel (PLAF)**.
  - Because each Swing component is **rendered by Java code** not by native peers, the look and feel of a component is under the control of Swing.
- It is possible to separate the **look and feel of a component** from the **logic of the component.**
  - ❑ Advantage:
  - It is possible to change the way that a component is rendered *without affecting any of its other aspects*
    - it is possible to **"plug in" a new look and feel for any given component** without creating any side effects in the code that uses that component.

Swing Key Features-**Swing Supports a Pluggable Look and Feel**

- It is possible to define entire sets of look-and-feels that represent *different GUI styles*
- To use a specific style, its look and feel is simply "plugged in."
  - Once this is done, all components are automatically rendered using that style.
- Pluggable look-and-feels offer several important advantages.
  - It is possible to define a look and feel that is **consistent across all platforms.**
  - it is possible to create a look and feel that acts like a specific platform.
    - For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
    - It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Difference between Swing and AWT

Swing	AWT
Swing components are **not platform-dependent.**	AWT components are **platform-dependent**
Swing provides several **additional components** such as scroll panes, trees etc in addition to other standard components	The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
Swing is **written entirely in Java**. So swing components are **light-weight**	AWT components use native code. So they are heavy weight
Swing supports a **pluggable look and feel** (PLAF) that can be dynamically changed at run-time depending on environment	In AWT look and feel of each component is fixed and it is difficult to change its look and feel.
Swing follow MVC	AWT does not follow MVC

Swing	Applet
• Swing is **light weight** Component.	• Applet is heavy weight Component.
• Swing needs **main method** to execute the program.	• Applet does not need main method to execute.
• Swing **follows MVC**(Model view Controller).	• Applet does not follow MVC.
• Swing have its own Layout like most popular **Box Layout.**	• Applet uses AWT Layouts like Flowlayout.
• Swing uses for stand alone Applications.	• Applet need HTML code for Run.
• To execute Swing, the browser is not needed.	• To execute Applet program we need browsers like Appletviewer, web browser etc.

Swing

- Swing is a set of program components for **Java** programmers that provide the **ability** to **create graphical user interface (GUI) components**, such as buttons and scroll bars, that are independent of the windowing system for specific operating system .

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 5 - **Graphical User Interface and Database support of Java**

(Part 2)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Swings**
  - ☑ MVC
  - ☑ Swing Controls
  - ☑ Components and Containers

MVC

- A visual component is a composite of three distinct aspects:
  - The **state information** associated with the component **MODEL**
  - The way that the **component looks** when rendered on the screen **VIEW**
  - The way that the **component reacts** to **CONTROLLER**

- MVC (Model View Controller) architecture is successful because **each piece of the design corresponds to an aspect of a component.**

MVC(contd.)

- In **MVC** terminology,
  - the ***Model*** corresponds to the **state information associated with the component**.
    - For example, in the case of a check box, the **model** contains a field that indicates if the box is checked or unchecked.
  - The ***View*** determines **how the component is displayed on the screen**, including any aspects of the view that are affected by the current state of the model.
  - The ***Controller*** determines **how the component reacts to the user**.
    - For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). So view changes.

MVC(contd.)

- By separating a component into a model, a view, and a controller, the *specific implementation of each can be changed* **without affecting the other two**.
- Swing uses a **modified version of MVC** that **combines the view and the controller into a single logical entity** called the ***UI delegate***.
  - So, Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture.
- Swing's pluggable look and feel is made possible by its Model-Delegate architecture.
- Because the view (look) and controller (feel) are ***separate*** from the model, the look and feel can be changed without affecting how the component is used within a program.

MVC(contd.)

- To support the **Model-Delegate** architecture, most Swing components contain two objects.
  - The first represents the **model**.
    - Models are defined by interfaces
  - The second represents the **UI delegate**.
    - UI delegates are classes that inherit **ComponentUI.**

Swing Controls

- Swing controls plays major role in swing applications, every application will have some components and their corresponding event listener.
- Swing controls will inherit the properties from following classes.
  - Component
  - Container
  - JComponent.

Swing Controls

- Some swing controls are
  - Container Control(Parent control)
    - It hold other controls(child controls or components)
      - JFrame, JPanel
  - Child Control –These controls are inside container control
    - They can only exist inside container control. They can be components or containers.
      - JTextArea,JLabel, JButton
- When container control is deleted then its child controls are also deleted.

Components and Containers

- A Swing GUI consists of two key items: ***components*** and ***containers***
- A container holds a group of components.
  - So a **container** is a **special type of component** that is designed to hold other components.
- To display a component, it must be held within a container.
- So all Swing GUIs will have at least one container.
- Because containers are components, a *container can also hold other containers*.

Components

- Swing components are derived from the **JComponent class.**
- **JComponent** provides the ***functionality*** that is common to all components.
  - E.g **JComponent** supports the pluggable look and feel.
- **JComponent** inherits the AWT classes **Container** and **Component**..
  - So, a Swing component is **built on and compatible** with an AWT component.

Components(contd.)

- All of Swing's **components** are represented by **classes** defined within the package **javax.swing.**
- The class names for Swing components are

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

** All component classes begin with the letter **J.***

Containers

- Swing defines **two types** of containers.
  - The first are **top-level containers(heavyweight)**
  - The second type of containers supported by Swing are **lightweight containers**.

Containers(contd.)

- The first are **top-level containers**
  - **JFrame, JApplet, JWindow, and JDialog.**
  - These containers **do not** inherit **JComponent**
  - They inherit the AWT classes **Component and Container**
  - The top-level containers are **heavyweight.**
  - A top-level container must be at the top of a containment hierarchy
  - A top-level container is not contained within any other container.
  - Every containment hierarchy must begin with a top-level container.
  - The one most commonly used for applications is **JFrame.**
  - The one used for applets **is JApplet.**

Containers (contd.)

- The second type of containers supported by Swing are **lightweight containers**.
  - Lightweight containers *do* ***inherit JComponent.***
    - E.g. **JPanel** is a general-purpose container.
  - Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.
  - We can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

Top-level containers

- Top level containers **do not** inherit **Jcomponent.**
- **Heavyweight**
- A top-level container is not contained within any other container.

- E.g. **JFrame, JApplet, JWindow, and JDialog**

Lightweight containers.

- Lightweight containers *do inherit **JComponent.***
- **Lightweight**
- A lightweight container can be contained within another container.

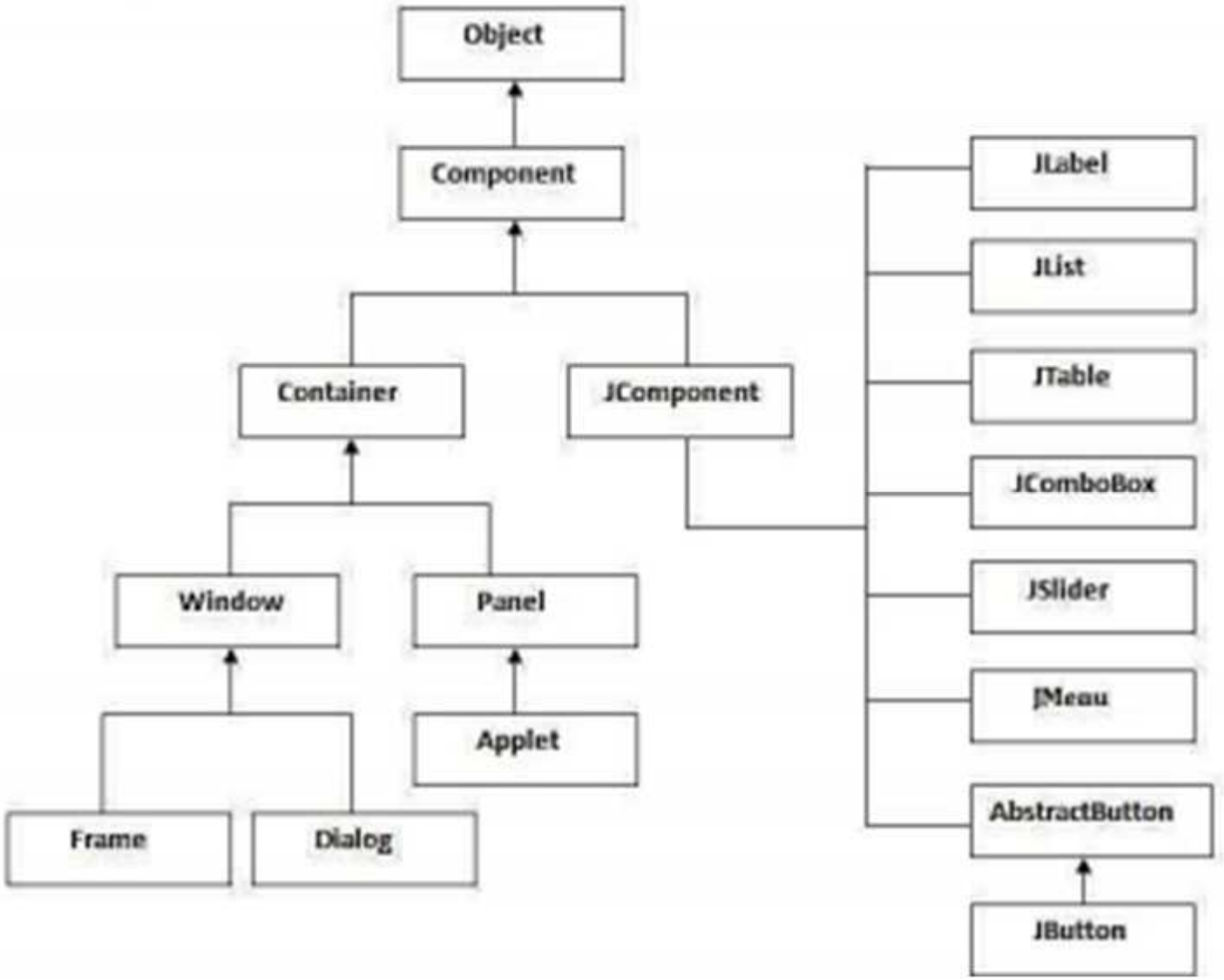
- E.g. **JPanel**

The Top-Level Container Panes

- The **Top-Level Container Panes**
  - Each top-level container defines a set of panes.
  - At the **top** of the hierarchy is an instance of **JRootPane**.
    - JRootPane is a lightweight container whose purpose is to manage the other panes.
    - It also helps manage the optional menu bar.
    - The panes that comprise the root pane are called
      - the **glass pane,**
      - **the content pane,**
      - **the layered pane**.

The Top-Level Container Panes

- Glass pane :
  - The glass pane is the **top-level pane.**
  - It sits above and completely covers all other panes.
  - By default, it is a **transparent** instance of JPanel.
- Layered pane :
  - The layered pane is an instance of JLayeredPane.
  - The layered pane allows components to be **given a depth value.**
  - This value determines which component overlays another.
- Content pane:
  - The pane with which your application will **interact the most** is the content pane
  - When we add a component, such as a button, to a top- level container, **we will add it to the content pane**.
  - By default, the content pane is an **opaque** instance of JPanel.


Object
Component
Container
JComponent
Window
Panel
Applet
Frame
Dialog
JLabel
JList
JTable
JComboBox
JSlider
JMenu
AbstractButton
JButton

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 5 - **Graphical User Interface and Database support of Java**

(Part 3)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Swings**
  - ☑ Swing Packages
  - ☑ Event Handling in Swings.

Swing Packages

- Swing is a very large subsystem and makes use of many packages.
  - These are the **packages** used by Swing that are **defined by Java SE 6**.
- The main package is **javax.swing.**
  - This package **must be imported into any program that uses Swing.**
  - It contains the **classes that implement the basic Swing components**, such as push buttons, labels, and check boxes.

Swing packages(contd.)

javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

A Simple Swing Application

- There are two types of Java programs in which Swing is typically used.
  1. desktop application.
  2. applet

A Simple Swing Application

- **Q.** Write a swing program that uses two Swing components: **Jframe** and **JLabel.** The program uses a JFrame container to hold an instance of a JLabel. The label displays a short text message

- **JFrame** is the top-level container that is commonly used for Swing applications. **JLabel** is the Swing component that creates a label, which is a component that displays information. **The label is Swing's simplest component because it is passive.**

  - That is, a label does not respond to user input. It just displays output.

```
import javax.swing.*;
class SwingDemo
{
    SwingDemo()
    {
        // Create a new JFrame container. With title- A Simple Swing
        JFrame jfrm = new JFrame("A Simple Swing ");

        // Give the frame an initial size. Width=275 height =100
        jfrm.setSize(275, 100);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Create a text-based label
        JLabel jlab = new JLabel(" Swing is powerful GUI");
        // Add the label to the content pane.
        jfrm.add(jlab);
        // Display the frame.
        jfrm.setVisible(true);
    }
    public static void main(String args[])
    {
        // Create the frame on the event dispatching thread.
        SwingUtilities.invokeLater( new Runnable()
                                        {
                                            public void run()
                                             {
                                            new SwingDemo();
                                             }
                                        }
                                    );
    }   }
```

Too compile this program,
javac SwingDemo.java
To run the program,
java SwingDemo

```
// A simple Swing application.
import javax.swing.*;
class SwingDemo {
SwingDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("A Simple Swing ");
// Give the frame an initial size.
jfrm.setSize(275, 100);
// Terminate the program when the user closes
    the application.
jfrm.setDefaultCloseOperation(JFrame.EXI
    T_ON_CLOSE);
// Create a text-based label
JLabel jlab = new JLabel(" Swing is
    powerful GUI");
// Add the label to the content pane.
jfrm.add(jlab);
// Display the frame.
jfrm.setVisible(true);
}
public static void main(String args[])
{
// Create the frame on the event
dispatching thread.
SwingUtilities.invokeLater(new
    Runnable() {
public void run() {
new SwingDemo();
}
});
}
}
```

- **javax.swing** defines classes that implement labels, buttons, text controls, and menus.
- The constructor is where most of the action of the program occurs. It begins by creating a **JFrame,** using this line of code:

JFrame jfrm = new JFrame("A Simple Swing ");

- This creates a container called **jfrm** that defines a rectangular window complete with a **title bar; close, minimize, maximize, and restore buttons; and a system menu.**
- Thus, it creates a standard, top-level window.
- The **title of the window** is passed to the constructor
  - Here title is *A Simple Swing*

- The window is sized using this statement:

jfrm.setSize(275, 100);

- The **setSize( )** method (which is inherited by JFrame from the AWT class Component) sets the dimensions of the window, which are specified in pixels. Its general form :

void **setSize**(in t *width, int height)*

- We want the **entire application to terminate when its top-level window is closed.** There are a couple of ways to achieve this. The easiest way is to call **setDefaultCloseOperation( ),**

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

- After this call executes, closing the window causes the entire application to terminate.

- The general form of **setDefaultCloseOperation( ) is :**

void **setDefaultCloseOperation**(int ***what)***

  - The value passed in ***what*** determines ***what happens when the window is closed.***

- *There are* several options :

  JFrame.EXIT_ON_CLOSE

  JFrame.DISPOSE_ON_CLOSE

  JFrame.HIDE_ON_CLOSE

  JFrame.DO_NOTHING_ON_CLOSE

- The next line of code creates a Swing **JLabel component:**

JLabel jlab = new **JLabel**(" Swing is powerful GUI");

- The next line of code **adds the label to** the content pane of the frame:

  jfrm.**add**(jlab);

- Thus, to add a component to a frame, we must add it to the frame's content pane. This is accomplished by calling **add( ) on the JFrame reference (jfrm in this case). The** general form of **add( ) is:**

Component **add**(Component *comp)*

- The content pane can be obtained by calling getContentPane( ) on a JFrame instance

```
Container getContentPane( )
```

- The last statement in the **SwingDemo constructor causes the window to become visible:**

```
jfrm.setVisible(true);
```

- SwingDemo constructor is invoked using the following lines of code:

```
SwingUtilities.invokeLater(   new Runnable() {
                                                    public void run()
                                                    {
                                                    new SwingDemo();
                                                     }
                                                    }
                                    );
```

- This sequence causes a **SwingDemo object to be created on the *event dispatching thread*** rather than on the main thread of the application.
- Swing programs are event-driven.

- To enable **the GUI code to be created on the event dispatching thread**, we must use one of two methods that are defined by the **SwingUtilities class**.
- These methods are
  - **invokeLater( )**
  - **invokeAndWait( ).**

static void **invokeLater**(Runnable *obj)*

static void **invokeAndWait**(Runnable *obj)*

throws InterruptedException, InvocationTargetException

- The difference between the two methods is that
  - **invokeLater() returns immediately,**
  - but **invokeAndWait( ) waits until obj.run( ) returns**

Event Handling in Swings

- **Delegation event model** is the event handling mechanism used by Swing.
- Swing uses the same events as does the AWT, and these events are packaged in **java.awt.event.**
- Events specific to Swing are stored in **javax.swing.event**

Swing-Event handling E.g.

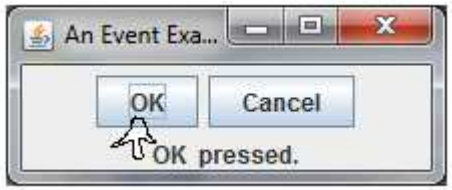
- Q. Write a program in swing to create a frame with title “An Event Example ”.
  - Give FlowLayout to frame and set a width =220 and height=90
  - Frame has two buttons Ok and Cancel.
  - Frame has a label that display the message “Push a button”.
  - When we click the OK button it prints the message “OK pressed” in the label.
  - When we click the Cancel button it prints the message “Cancel pressed” in the label

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo extends JFrame implements ActionListener
{   JLabel jlab;
    EventDemo()
    {               // Create a new JFrame container.
        JFrame jfrm = new JFrame("An Event Example");
                    // Specify FlowLayout for the layout manager.
        jfrm.setLayout(new FlowLayout());
                    // Give the frame an initial size.
        jfrm.setSize(220, 90);
                    // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    // Make two buttons.
        JButton jbtnOk = new JButton("OK");
        JButton jbtnCancel = new JButton("Cancel");
```

```
    // Add action listener for Ok button.
jbtnOk.addActionListener(new ActionListener()
                                {
                                        public void actionPerformed(ActionEvent ae)
                                         {
                                        jlab.setText("OK  pressed.");
                                        }
                                  }
                                );
    // Add action listener for Cancel button.
jbtnCancel.addActionListener(new ActionListener()
                                {
                                        public void actionPerformed(ActionEvent  ae)
                                        {
                                                jlab.setText("Cancel  pressed.");
                                        }
                                }
                                );
```

```
        // Add the buttons to the content pane.
        jfrm.add(jbtnOk);
        jfrm.add(jbtnCancel);
        // Create a text-based label.
        jlab = new JLabel("Press a button.");
        // Add the label to the content pane.
        jfrm.add(jlab);
        // Display the frame.
        jfrm.setVisible(true);
}
```

```
    public static void main(String args[])
    {
        // Create the frame on the event dispatching thread.
        SwingUtilities.invokeLater(new Runnable()
                                    {
                                        public void run()
                                        {
                                        new EventDemo();
                                        }
                                    }
                                );
    }
}
```



- The java.awt package is needed because
  - it contains the FlowLayout class, which supports the standard flow layout manager used to lay out components in a frame
  - It defines the ActionListener interface and the ActionEvent class.
- The **EventDemo** constructor begins by creating a JFrame called jfrm with title -An Event Example

```
JFrame jfrm = new JFrame("An Event Example");
```

- It then sets thelayout manager for the content pane of jfrm to FlowLayout.

```
jfrm.setLayout(new FlowLayout());
```

- By **default**, the content pane uses **BorderLayout** as its layout manager.

- After setting the size and default close operation, **EventDemo() creates two push buttons**, as shown here:

JButton jbtnOk = new **JButton**(“Ok");

JButton jbtnCancel = new **JButton**(”Cancel");

  - The first button will contain the text “Ok” and the second will contain the text “Cancel”.

- When a push button is pressed, it generates an **ActionEvent.**

- Thus, JButton provides the addActionListener( ) method, which is used to add an action listener so that button will respond to events. (JButton also provides removeActionListener( ) to remove a listener)

- event listeners for the button's action events are added by the code shown belo:
- // Add action listener for Ok button.

```
jbtnOK.addActionListener(new ActionListener()
                                {
                                public void actionPerformed(ActionEvent ae)
                                {  jlab.setText("OKwas pressed.");
                                }
                                });
```

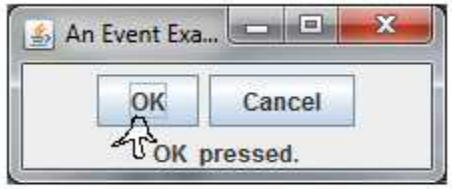
This can also be written as:

```
jbtnOK.addActionListener(this);
……
}
public void actionPerformed(ActionEvent ae)
{    String s = ae.getActionCommand();      //to get the name written in button
            if(s.equalsIgnoreCase("ok"))        //to have case insensitive comparison
                    jlab.setText("OK  pressed.");
}
```

Simple Program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemoSwing  extends JFrame implements ActionListener
{   JLabel jlab;
    EventDemoSwing()
    {       JFrame jfrm = new JFrame("An Event Example");
            jfrm.setLayout(new FlowLayout());
            jfrm.setSize(220, 90);
            jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            JButton jbtnOk = new JButton("OK");
            JButton jbtnCancel = new JButton("Cancel");
            jbtnOk.setToolTipText("click");
            jbtnOk.addActionListener(this);
            jbtnCancel.addActionListener(this);
            jfrm.add(jbtnOk);
            jfrm.add(jbtnCancel);
            jlab = new JLabel("Press a button.");
            jfrm.add(jlab);
            jfrm.setVisible(true);
    }
```

```
public void actionPerformed(ActionEvent ae)
    {       //store the name written in button that is clicked ,in variable s
            String s = ae.getActionCommand();
            if(s.equalsIgnoreCase("ok"))
                    jlab.setText("OK  pressed.");
            else if(s.equalsIgnoreCase("cancel"))
                    jlab.setText("Cancel  pressed.");
    }
    public static void main(String args[])
    {       SwingUtilities.invokeLater(new Runnable()
                                    {       public void run()
                                            {
                                            new EventDemoSwing();
                                            }
                                    }
                                    );
    }
}
```



Create a Swing Applet

- A Swing applet extends **Japplet.**
  - **JApplet** is derived from **Applet.**
- Swing applets use the same four lifecycle methods as Applet
- **init( ),**
- **start( ), stop( ), and destroy( ).**
- Swing applet will not normally override the **paint( ) method**

- Write a program using **SWING APPLET**
  - It should have two buttons Ok and Cancel.
  - Label to display message “Push a button”.
  - When we click the OK button it prints the message “OK pressed” in the label.
  - When we click the Cancel button it prints the message “Cancel pressed” in the label

```
// A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/*
This HTML can be used to launch the applet:
    <object code="MySwingApplet" width=220 height=90>
    </object>
*/
    public class MySwingApplet extends JApplet implements ActionListener
    {
        JButton jbtnOk;
        JButton jbtnCancel;
        JLabel jlab;
```

```
// Initialize the applet.
        public void init() {
        try {       SwingUtilities.invokeAndWait(new Runnable ()
                                        {
                                        public void run() {
                                        makeGUI(); // initialize the GUI
                                        }
                                        });
                    } catch(Exception exc)
                    {
                    System.out.println("Can't create because of "+ exc); }
                    }
```

```
    private void makeGUI()
{   // Set the applet to use flow layout.
        setLayout(new FlowLayout());
        // Make two buttons.
        jbtnOk = new JButton("Ok");
        jbtnCancel = new JButton("Cancel");
        // Add action listener for ok.
        jbtnOk.addActionListener(this);
        // Add action listener for Cancel.
        jbtnCancel.addActionListener(this);
        // Add the buttons to the content pane.
        add(jbtnOk);
        add(jbtnCancel);
        // Create a text-based label.
        jlab = new JLabel("Press a button.");
        // Add the label to the content pane.
        add(jlab);
        }
```

```
public void actionPerformed(ActionEvent ae)
  {
        String s = ae.getActionCommand();
              if(s.equalsIgnoreCase("Ok"))
                    jlab.setText("Ok was pressed.");
              else if(s.equalsIgnoreCase("Cancel"))
                    jlab.setText("Cancel was pressed.");
    }
}
```

COMPILE
javac MySwingApplet.java

RUN
appletviewer MySwingApplet.java

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 5 - **Graphical User Interface and Database support of Java**

(Part 4)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Swings**
  - ☑ Swing Layout Managers

Swing Layout Managers

- A layout manager **automatically arranges our controls within a window** by using some type of algorithm.
- Each Container object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the LayoutManager interface.
- The layout manager is set by the **setLayout( )** method.
  - If no call to setLayout( ) is made, then the default layout manager is used.
- The **setLayout( )** method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Swing Layout Managers(contd.)

- The *layout manager is notified* each time we add a component to a container.
- Each layout manager keeps track of a list of components that are stored by their names.
- Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize**( ) and **preferredLayoutSize**( ) methods.
  - Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods.

- Java has several predefined **LayoutManager classes,**
  - **FlowLayout**
  - **BorderLayout**
  - **GridLayout**
  - **CardLayout**
  - **GridBagLayout**

FlowLayout

- The direction of the layout is governed by the container's component orientation property, which, by default, is **left to right, top to bottom.**
- In low Layout
  - components are laid out **line-by-line** **beginning at the upper-left corner.**
  - When a line is filled, layout advances to **the next line**.
  - A small space is left between each component, above and below, as well as left and right.
- The constructors for **FlowLayout:**

  FlowLayout( )

  FlowLayout(int *how)*

  FlowLayout(int *how, int horz, int vert)*

FlowLayout(contd.)

- FlowLayout( ) creates the default layout, which **centers components** and leaves five pixels of space between each component.
- FlowLayout(int *how)* lets us specify **how each line is aligned**.
  - Valid values for *how are as follows:*
    - FlowLayout.LEFT
    - FlowLayout.CENTER
    - FlowLayout.RIGHT
    - FlowLayout.LEADING
    - FlowLayout.TRAILING
      - These values specify left, center, right, leading edge, and trailing edge alignment, respectively.
- FlowLayout(int *how, int horz, int vert)* allows us to specify the **horizontal and vertical space left between components** in *horz and vert,* respectively.

```
import java.awt.*;      import java.awt.event.*;
import javax.swing.*;
class EventDemoSwing  extends JFrame implements
      ActionListener{
      JLabel jlab;
      JFrame jfrm;
      JButton jbtnOk;
      JButton jbtnCancel;
      EventDemoSwing()
      {
jfrm = new JFrame("An Event Eg");
jfrm.setLayout(new FlowLayout());
jfrm.setSize(220, 90);
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jbtnOk = new JButton("OK");
JButton jbtnCancel = new JButton("Cancel");
jbtnOk.setToolTipText("click");
jbtnOk.addActionListener(this);
jbtnCancel.addActionListener(this);
jfrm.add(jbtnOk);
jfrm.add(jbtnCancel);
jlab = new JLabel("Press a button.");
jfrm.add(jlab);
jfrm.setVisible(true);            }
```

```
public void actionPerformed(ActionEvent ae)
    {
    String s = ae.getActionCommand();
    if(s.equalsIgnoreCase("ok"))
          jlab.setText("OK  pressed.");
    else if(s.equalsIgnoreCase("cancel"))
          jlab.setText("Cancel  pressed.");
    }
          public static void main(String args[])
{SwingUtilities.invokeLater(new Runnable()
          {
          public void run()
          { new EventDemoSwing();
          }
          } );
  }   }
```

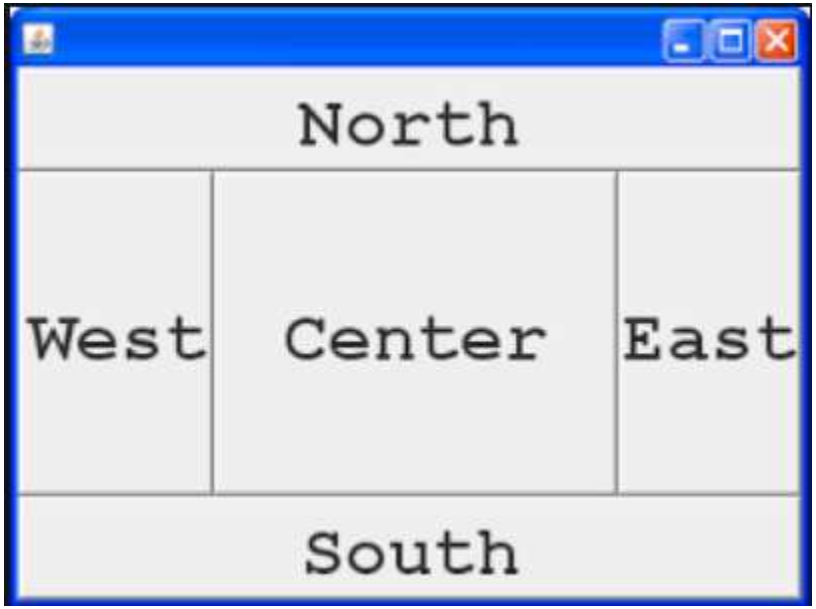
FlowLayout(contd.)

If in code we modify the layout as

```
jfrm.setLayout(new FlowLayout(FlowLayout.LEFT));
```

BorderLayout

- By default, the content pane associated with a **JFrame uses border layout.**
- The **BorderLayout class implements a common layout** style for top-level windows.
- It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as
  - North,
  - South
  - East
  - West.
- The middle area is called
  - Center



BorderLayout(contd.)

- The constructors defined by **BorderLayout:**

BorderLayout( )

- BorderLayout( ) creates a default border layout.
- BorderLayout(int *horz, int vert)* specify the horizontal and vertical space left between components in *horz and vert, respectively*

BorderLayout(contd.)

- BorderLayout defines the following constants
  - BorderLayout.CENTER
  - BorderLayout.SOUTH
  - BorderLayout.EAST
  - BorderLayout.WEST
  - BorderLayout.NORTH
- When adding components, you will use these constants with the following form of**add( ),** which is defined by Container**:**

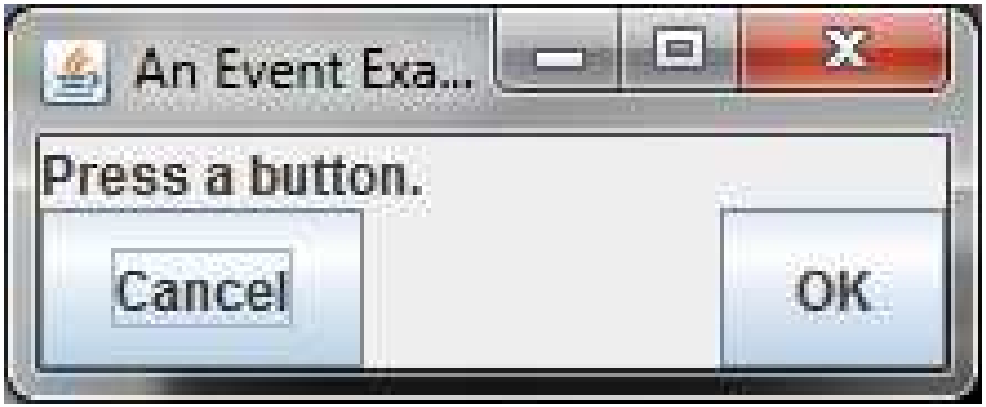
void **add**(Component *compObj, Object region)*

  - Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

```
import java.awt.*;      import java.awt.event.*;
import javax.swing.*;
class EventDemoSwing  extends JFrame implements
      ActionListener{
      JLabel jlab;
      JFrame jfrm;
      JButton jbtnOk;
      JButton jbtnCancel;
      EventDemoSwing()
      {
jfrm = new JFrame("An Event Eg");
jfrm.setLayout(new BorderLayout ());
jfrm.setSize(220, 90);
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jbtnOk = new JButton("OK");
JButton jbtnCancel = new JButton("Cancel");
jbtnOk.setToolTipText("click");
jbtnOk.addActionListener(this);
jbtnCancel.addActionListener(this);
jfrm.add(jbtnOk,BorderLayout.EAST);
jfrm.add(jbtnCancel, BorderLayout.WEST);
jlab = new JLabel("Press a button.");
jfrm.add(jlab BorderLayout.NORTH);
jfrm.setVisible(true);            }
```

```
public void actionPerformed(ActionEvent ae)
    {
    String s = ae.getActionCommand();
    if(s.equalsIgnoreCase("ok"))
          jlab.setText("OK  pressed.");
    else if(s.equalsIgnoreCase("cancel"))
          jlab.setText("Cancel  pressed.");
    }

          public static void main(String args[])
{SwingUtilities.invokeLater(new Runnable()
          {
          public void run()
          { new EventDemoSwing();
          }
          } );
    }   }
```

BorderLayout(contd.)



Using Insets

- Sometimes we may want to **leave a small amount of space** between the container that holds the components and the window that contains it.
  - To do this, override the **getInsets( ) method** that is defined by **Container.**
  - **getInsets( ) method** returns an Insets object that contains the top, bottom, left, and right inset to be used when the container is displayed.
  - These values are used by the layout manager to inset the components when it lays out the window
- The constructor for **Insets is shown here:**

  Insets(int *top, int left, int bottom, int right)*

  - The values passed in top, left, bottom, and right specify the *amount of space between the container and its enclosing window*

Using Insets(contd.)

- The **getInsets( ) method has this general form:**

  Insets **getInsets**( )

  – When overriding this method, we must return a new **Insets object** that contains the inset spacing you desire.

GridLayout

- **GridLayout** lays out components in a **two-dimensional grid**.
  - We can define the number of rows and columns.
- The constructors supported by GridLayout are

GridLayout( )

GridLayout(int *numRows, int numColumns)*

GridLayolayoutut(int *numRows, int numColumns, int horz, int vert)*

  - GridLayout( ) creates a single-column grid
  - GridLayout(int *numRows, int numColumns)* creates a grid layout with the specified number of rows and columns.
  - GridLayolayoutut(int *numRows, int numColumns, int horz, int vert)* allows us to specify the horizontal and vertical space left between components in *horz and vert*, respectively. Either *numRows* or *numColumns* can be zero. *Specifying numRows* as zero allows for unlimited-length columns.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class GridDemo  extends JFrame implements  ActionListener
{    JLabel jlab;
     JFrame jfrm;
     GridDemo()
     { jfrm = new JFrame("An Event Eg");
     jfrm.setLayout(new GridLayout (2,2));
     jfrm.setSize(220, 90);
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     JButton jbtnOne = new JButton("One");
     JButton jbtnTwo= new JButton("Two");
     JButton jbtnThree = new JButton("Three");
     JButton jbtnFour = new JButton("Four");
     jbtnOne.addActionListener(this);
     jbtnTwo.addActionListener(this);
     jbtnThree.addActionListener(this);
     jbtnFour.addActionListener(this);
     jfrm.add(jbtnOne);
     jfrm.add(jbtnTwo);
     jfrm.add(jbtnThree);
     jfrm.add(jbtnFour);
     jfrm.setVisible(true);
     }
public void actionPerformed(ActionEvent ae)
    {       String s = ae.getActionCommand();
            if(s.equalsIgnoreCase("one"))
                    jlab.setText("One  pressed.");
            else if(s.equalsIgnoreCase("two"))
 jlab.setText("Two pressed.");
            else if(s.equalsIgnoreCase("three"))
            jlab.setText("Three pressed.");
            else if(s.equalsIgnoreCase("four"))
            jlab.setText("Four pressed.");
     }
     public static void main(String args[])
    {SwingUtilities.invokeLater(new Runnable()
            { public void run()
            { new GridDemo(); }                    }
     );
     }
}
```

jfrm.setLayout(new GridLayout (2,2));

- This set 2 rows and 2 columns
- Components are filled in order from first row first column

(0,0) (0,1)

(1,0) (1,1)

CardLayout

- The CardLayout class is unique among the other layout managers in that it **stores several different layouts**.
- Each layout can be thought of as being on **a separate index card in a deck** that **can be shuffled** so that **any card is on top at a given time**
- This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.
- **CardLayout** provides these two constructors:

CardLayout( )

CardLayout(int *horz, int vert)*

  - CardLayout( ) creates a default card layout.
  - CardLayout(int *horz, int vert)* allows to specify the horizontal and vertical space left between components in *horz and vert, respectively.*

CardLayout(contd.)

- The cards are typically held in an object of type **Panel**.
- This panel must have **CardLayout** selected as its layout manager.
- The cards that form the deck are also typically objects of type **Panel**

CardLayout(contd.)

- We must **create**
  - **a panel** that contains the deck and
  - a **panel** for each card in the deck.
- Next, we **add components that form each card** to the appropriate panel.
- We then **add these panels to the panel** for which CardLayout is the layout manager.
- Finally, we **add this panel to the window.**

- Once these steps are complete, we must provide some way for the user to select between cards.
  - One common approach is to include one push button for each card in the deck.

CardLayout(contd.)

- Use **add( )** when adding cards to a panel:

void **add**(Component *panelObj, Object name)*

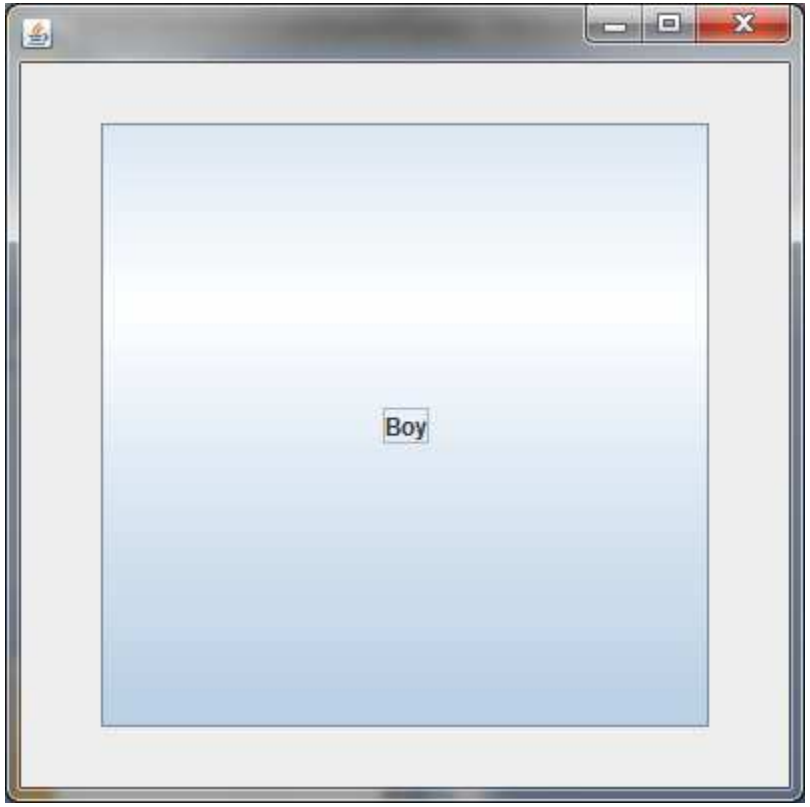
  - Here, *name is a string that specifies the name of the card whose panel is specified by panelObj.*

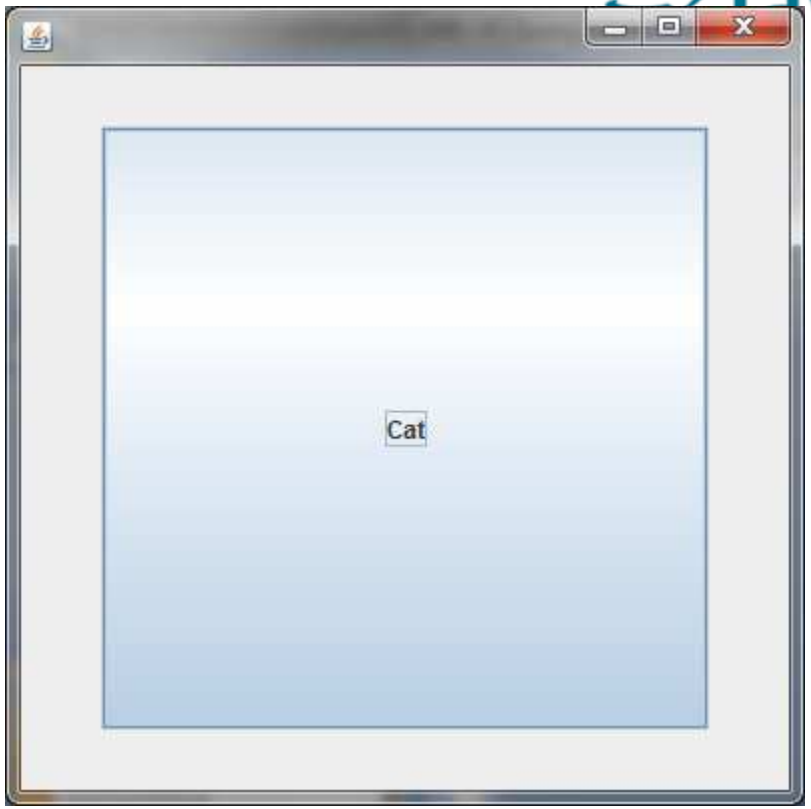
- After we have created a deck, our program activates a card by calling one of the following methods defined by **CardLayout:**

  void first(Container *deck)*
  void last(Container *deck)*
  void next(Container *deck)*
  void previous(Container *deck)*
  void show(Container *deck, String cardName)*

  - Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card.
  - Calling first( ) causes the first card in the deck to be shown. To show the last card, call last( ). To show the next card, call next( ). To show the previous card, call previous( ). Both next( ) and previous( ) automatically cycle back to the top or bottom of the deck, respectively. The show( ) method displays the card whose name is passed in *cardName*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutDemo extends JFrame implements
     ActionListener
{
CardLayout card;
JButton b1,b2,b3;
Container c;
   CardLayoutDemo()
     {
     c=getContentPane();
      card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
       c.setLayout(card);
       b1=new JButton("Apple");
       b2=new JButton("Boy");
       b3=new JButton("Cat");
       b1.addActionListener(this);
       b2.addActionListener(this);
       b3.addActionListener(this);

       c.add(b1);c.add(b2);c.add(b3);

   }
 public void actionPerformed(ActionEvent e)
     {
    card.next(c);
     }
public static void main(String[] args)
     {
   CardLayoutDemo cl=new  CardLayoutDemo();
   cl.setSize(400,400);
   cl.setVisible(true);
cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
     }
 }
```





- When we click the button it changes to next button like cards

GridBagLayout

- We can specify the **relative placement of components by specifying their positions** within cells inside a grid using **GridBagLayout**.
- The key to the grid bag is that each component can be a **different size**, and each **row** in the grid can have a **different number of columns**.
  - This is why the layout is called a *grid bag*.
- It's a **collection of small grids joined together**.
- The location and size of each component in a grid bag are determined by a set of **constraints** linked to it.
- The **constraints** are contained in an object of type **GridBagConstraints.**
  - **Constraints** include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

GridBagLayout(contd.)

- The general procedure for using a grid bag is to
  - first create a new **GridBagLayout object** and to make it the current layout manager.
  - Then, **set the constraints** that apply to each component that will be added to the grid bag.
  - Finally, add the components to the layout manager.
- **GridBagLayout** defines only one constructor

  GridBagLayout( )
- GridBagLayout defines several methods, of which many are protected and not for general use.
- One method is setConstraints( )

void **setConstraints**(Component *comp, GridBagConstraints cons)*

GridBagLayout(contd.)

GridBagConstraints defines several fields that to govern the size, placement, and spacing of a component.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is **GridBagConstraints.CENTER**.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are **GridBagConstraints.NONE** (the default), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL**, **GridBagConstraints.BOTH**.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**.
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**.
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

GridBagLayout(contd.)

- GridBagConstraints also defines several static fields that contain standard constraint values, such as
  - GridBagConstraints.CENTER
  - GridBagConstraints.VERTICAL
- When a component is smaller than its cell, you can use the anchor field to specify where within the cell the component's top-left corner will be locate

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

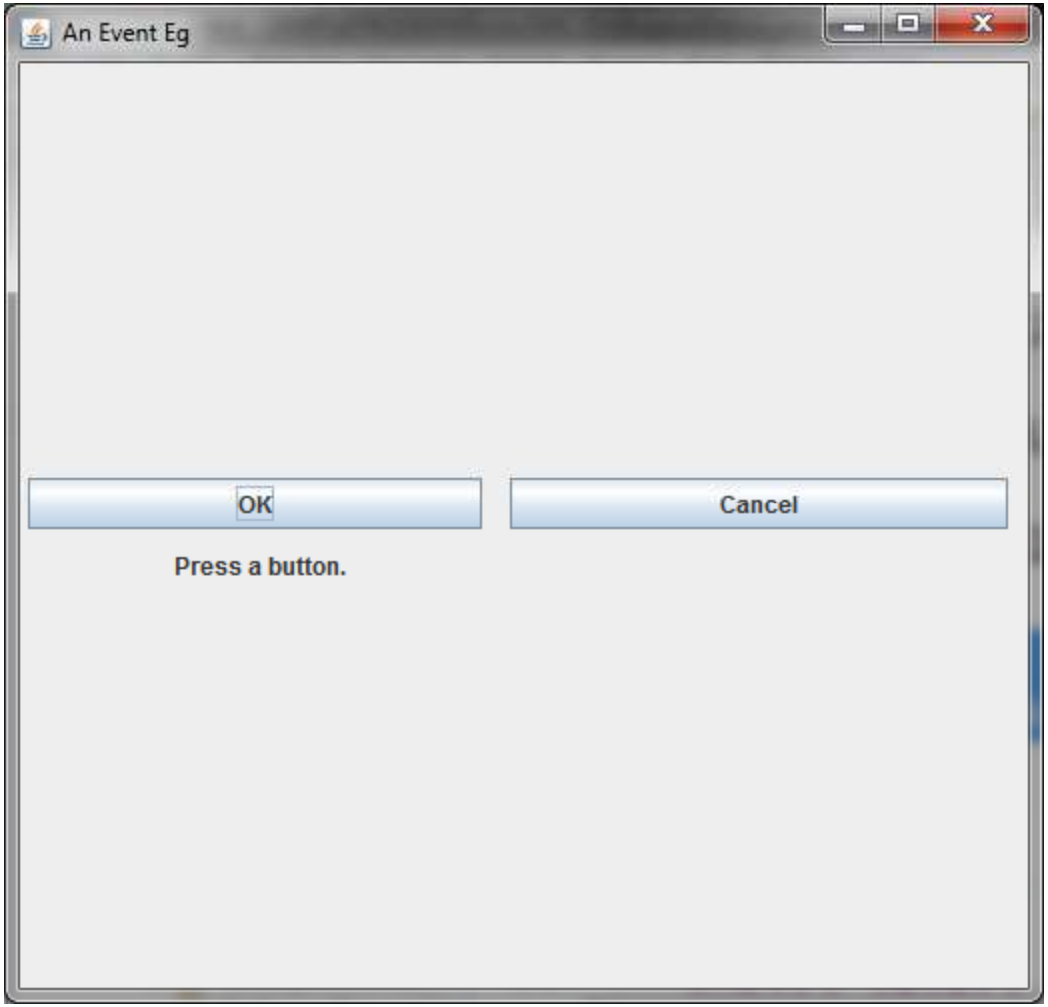
GridBagLayout(contd.)

- The second type of values that can be given to anchor is relative, which means the values are relative to the container's orientation,

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

```
import java.awt.*;      import java.awt.event.*;  import javax.swing.*;
public GridLayoutDemo() {
                    jfrm = new JFrame("An Event Eg");
          GridBagLayout gbag = new GridBagLayout();
          GridBagConstraints gbc = new GridBagConstraints();
                    jfrm.setLayout(gbag);
                    jfrm.setSize(520, 500);
                    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    jbtnOk = new JButton("OK");
                    jbtnCancel = new JButton("Cancel");
                    jbtnOk.setToolTipText("click");
                    jbtnOk.addActionListener(this);
                    jbtnCancel.addActionListener(this);
//Define the grid bag. // Use default row weight of 0 for first row.
          gbc.weightx = 1.0;                    // use a column weight of 1
          gbc.ipadx = 200;                      // pad by 200 units
          gbc.insets = new Insets(4, 4, 10, 10);  // inset slightly from top left
          gbc.anchor = GridBagConstraints.NORTH;
          gbc.gridwidth = GridBagConstraints.RELATIVE;
          gbag.setConstraints(jbtnOk, gbc);
          gbc.gridwidth = GridBagConstraints.REMAINDER;
          gbag.setConstraints(jbtnCancel, gbc);
```

```
                jlab = new JLabel("Press a button.");

                jfrm.add(jbtnOk);
                jfrm.add(jbtnCancel);
                jfrm.add(jlab);

                jfrm.setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String s = ae.getActionCommand();
        if(s.equalsIgnoreCase("ok"))
            jlab.setText("OK  pressed.");
        else if(s.equalsIgnoreCase("cancel"))
            jlab.setText("Cancel  pressed.");
    }

}
```


An Event Eg
OK
Cancel
Press a button.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 5 - **Graphical User Interface and Database support of Java**

(Part 5)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Swings**
  - ☑ Exploring Swings
    - ☑JFrame
    - ☑Jlabel
    - ☑The Swing Buttons
    - ☑JTextField

- Some of the swing components are:
  - JButton
  - JCheckBox
  - JComboBox
  - JLabel
  - JList
  - JRadioButton
  - JScrollPane
  - JTabbedPane
  - JTable
  - JTextField
  - JToggleButton
  - JTree

These components are all lightweight. They are all derived from **JComponent.**

JFrame

- Every containment hierarchy must begin with a top-level container.
- **JFrame** is a top level container that is commonly used for Swing applications.
- JFrame do not inherit **JComponent.**
- **JFrame** inherit the AWT classes **Component and Container.**
- The top-level containers are heavyweight.

JFrame(contd.)

JFrame jfrm = new **JFrame**("Swing Example);

- This creates a container called **jfrm** that
  - defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu.
  - Thus, it creates a standard, top-level window.
  - The title of the window is passed to the constructor.
    - Here it is ***Swing Example***

JFrame(contd.)

- The **setSize( )** method (which is inherited by JFrame from the AWT class Component) sets the dimensions of the window, which are specified in pixels.
- Its general form is :

void setSize(int *width, int height)*

E.g. **jfrm**.**setSize**(275, 100);

JFrame(contd.)

- If we want the entire application to terminate when its top-level window is closed the easiest way is to call **setDefaultCloseOperation( )**

E.g.

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

- After this call executes, closing the window causes the entire application to terminate.
- The general form of **setDefaultCloseOperation( ) is :**

  void **setDefaultCloseOperation**(int *what)*

  - *what* can be
    - JFrame.EXIT_ON_CLOSE
    - JFrame.DISPOSE_ON_CLOSE
    - JFrame.HIDE_ON_CLOSE
    - JFrame.DO_NOTHING_ON_CLOSE

JFrame(contd.)

- The content pane can be obtained by calling **getContentPane( )** on a JFrame instance.
- The **getContentPane( )** method is :

Container **getContentPane**( )

- The setVisible( ) method is inherited from the AWT Component class.
  - If its argument is true, the window will be displayed. Otherwise, it will be hidden.
    - By default, a JFrame is invisible, so setVisible(true) must be called to show it.
  - E.g.

    jfrm.**setVisible(true**);

JLabel

- JLabel is Swing's easiest-to-use component.
- It creates a label.
- **JLabel** can be used **to display text and/or an icon**.
- It is a **passive component** because it does not respond to user input.
- JLabel defines several constructors:

  JLabel(Icon *icon)*

  JLabel(String *str)*

  JLabel(String *str, Icon icon, int align)*

  – The *align argument specifies the* horizontal alignment of the text and/or icon within the dimensions of the label.
    - It must be one of the following values: **LEFT, RIGHT, CENTER, LEADING, or TRAILING**

JLabel(contd.)

- The easiest way to obtain an icon is to use the ImageIcon class.
- ImageIcon implements Icon and encapsulates an image.
- The following **ImageIcon** constructor obtains the image in the file named *filename*. the Icon parameter of JLabel's constructor

  ImageIcon(String *filename)*
- The icon and text associated with the label can be obtained by the following methods:

  Icon getIcon( )

  String getText( )
- The icon and text associated with a label can be set by these methods:

  void setIcon(Icon *icon)*

  void setText(String *str)*

The Swing Buttons

- Swing defines four types of buttons:

  JButton

  JToggleButton

  JCheckBox

  JRadioButton

- All are subclasses of the **AbstractButton class** (which extends JComponent)

The Swing Buttons(contd.)

- **AbstractButton** contains many methods that allow you to control the behavior of buttons.
- E.g. We can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover icon, which is displayed* when the mouse is positioned over a button.

  void setDisabledIcon(Icon *di)*

  void setPressedIcon(Icon *pi)*

  void setSelectedIcon(Icon *si)*

  void setRolloverIcon(Icon ri)

  – Here, *di, pi, si, and ri are the icons* for specific purpose

The Swing Buttons (contd.)

- We can get the text associated with a button using:

  String **getText**( )

- We can modify the text associated with a button using:

  void **setText**(String *str)*

- The model used by all buttons is defined by the **ButtonModel interface.**

JButton

- The **JButton** class provides the functionality of a **push button.**
- **JButton** allows an **icon, a string, or both** to be associated with the push button.
- Three of its constructors are shown here:

  JButton(Icon *icon)*

  JButton(String *str)*

  JButton(String *str, Icon icon)*

- When the button is pressed, an **ActionEvent** is generated**.**

JButton(contd.)

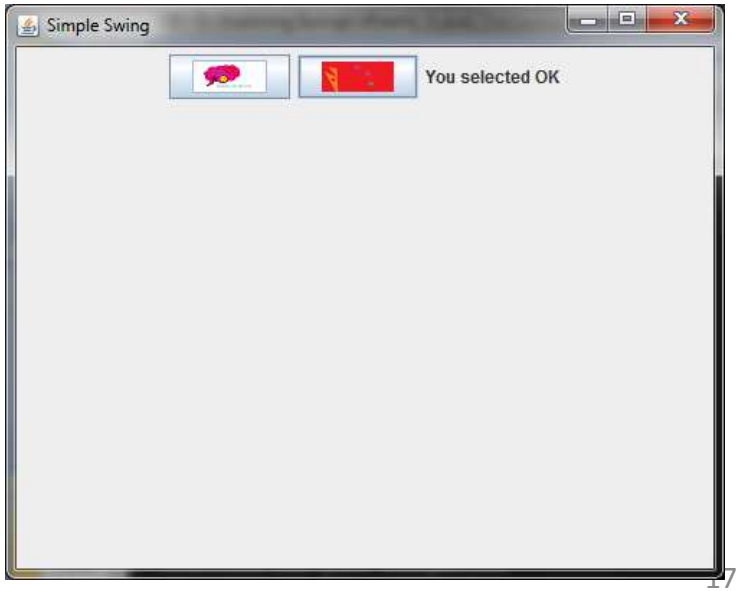
- Using the **ActionEvent** object passed to the **actionPerformed**( ) method of the registered **ActionListener,** we can obtain the *action command string associated with the button.*
- We can set the action command by calling **setActionCommand**( ) on the button.
- We can obtain the action command by calling **getActionCommand**( )

String **getActionCommand**( )

- The action command helps to identify the button.
  - Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class SwingButton extends
    JFrame implements
    ActionListener
{   JFrame jfrm;
    JButton jbok , jbcancel;
    JLabel jlab;
    SwingButton()
    {
jfrm = new JFrame("Simple
    Swing ");
jfrm.setSize(500, 400);
jfrm.setLayout(new
    FlowLayout());
jfrm.setDefaultCloseOperation(J
    Frame.EXIT_ON_CLOSE);
ImageIcon imgok= new
    ImageIcon("C:\\RJB\\image1.jpg");
    jbok = new JButton(imgok);
    jbok.setActionCommand("OK");
    jfrm.add(jbok);
ImageIcon imgcancel= new
    ImageIcon("C:\\RJB\\image2.jpg");
    jbcancel = new JButton(imgcancel);
jbcancel.setActionCommand("Cancel");
jbok.addActionListener(this);
jbcancel.addActionListener(this);
    jfrm.add(jbcancel);
jlab = new JLabel("Waiting button press ");
    jfrm.add(jlab);
    jfrm.setVisible(true);
    }
```

```
public void actionPerformed(ActionEvent ae)
    {
    jlab.setText("You selected " + ae.getActionCommand());
    }
    public static void main(String args[])
    {      SwingUtilities.invokeLater(new Runnable()
                                        {
                                                public void run()
                                                {
                                                new SwingButton ();
                                                }
                                        }
                                        );
    }
}
```



JToggleButton

- A toggle button looks just like a push button,
- It acts differently from push button because it has two states:
  - **Pushed**
  - **Released**
- When we press a toggle button, it **stays pressed.**
  - It does not pop back up as a regular push button.
- When we press the toggle button a second time, it releases (pops up).
- Each time a toggle button is pushed, it toggles between its two states
- Toggle buttons are objects of the **JToggleButton** class.

JToggleButton(contd.)

- JToggleButton implements **AbstractButton**.
- JToggleButton is a superclass for **JCheckBox** and **JRadioButton**
- **JToggleButton** defines several constructors.

JToggleButton(String *str)*

This creates a toggle button that contains the text passed in *str.*

- By **default**, the button is in the **off** position.
- **JToggleButton** uses a model defined by a nested class called JToggleButton .ToggleButtonModel.
- **JToggleButton** generates an action event each time it is pressed.
- When a **JToggleButton is pressed** in, it is **selected**.
- When it is popped out, it is **deselected**.

JToggleButton(contd.)

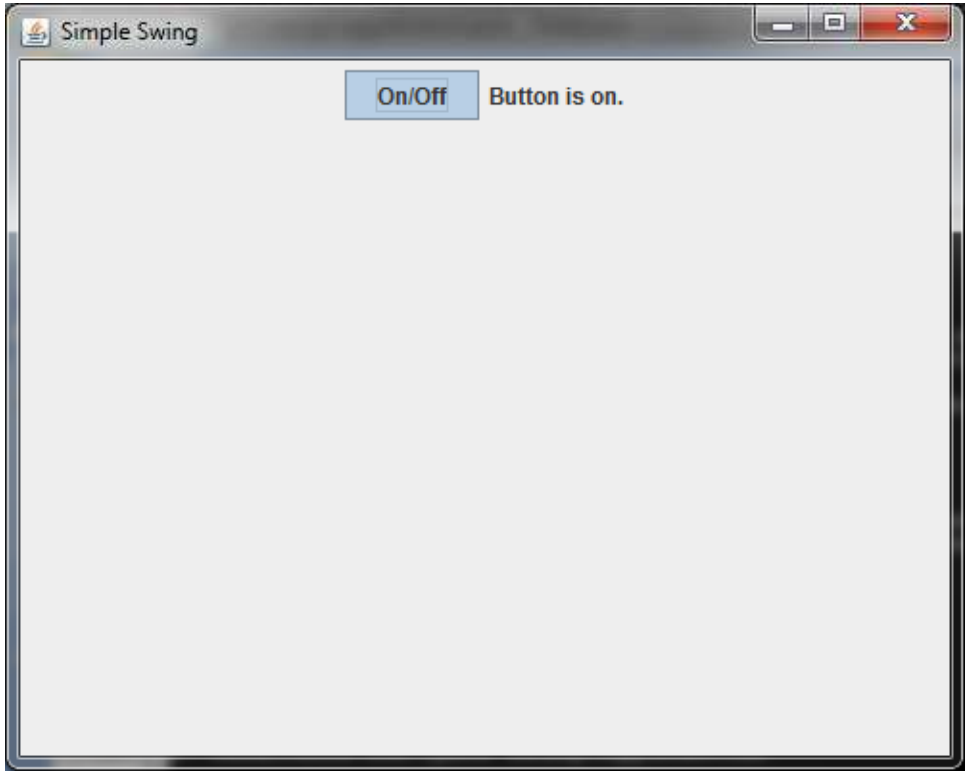
- To handle item events, we must implement the **ItemListener** interface.
- Each time an item event is generated, it is passed to the **itemStateChanged( )** method defined by **ItemListener.**
- Inside **itemStateChanged( ), the getItem( )** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event.

  Object **getItem( )**
- The easiest way to determine a toggle button's state is by calling the **isSelected( )** method.

  boolean **isSelected( )**
  - It returns **true if the button is selected** and false otherwise.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class SwingToggleButton extends
    JFrame implements ItemListener
{   JFrame jfrm;
    JToggleButton jtbn;
    JLabel jlab;
SwingToggleButton()
    {
jfrm = new JFrame("Simple Swing ");
jfrm.setSize(500, 400);
jfrm.setLayout(new FlowLayout());
jfrm.setDefaultCloseOperation(JFrame
    .EXIT_ON_CLOSE);
    jtbn=new JToggleButton("On/Off");
    jtbn.addItemListener(this);
    jfrm.add(jtbn);
    jlab = new JLabel("Button is OFF");
    jfrm.add(jlab);
    jfrm.setVisible(true);
    }
public void itemStateChanged(ItemEvent ie)
    {
        if(jtbn.isSelected())
            jlab.setText("Button is on.");
        else
            jlab.setText("Button is off.");
    }
```

```
    public static void main(String args[])
    {       SwingUtilities.invokeLater(new Runnable()
                                        {
                                                public void run()
                                                {
                                                new SwingToggleButton ();
                                                }
                                        }
                                        );
    }
}
```



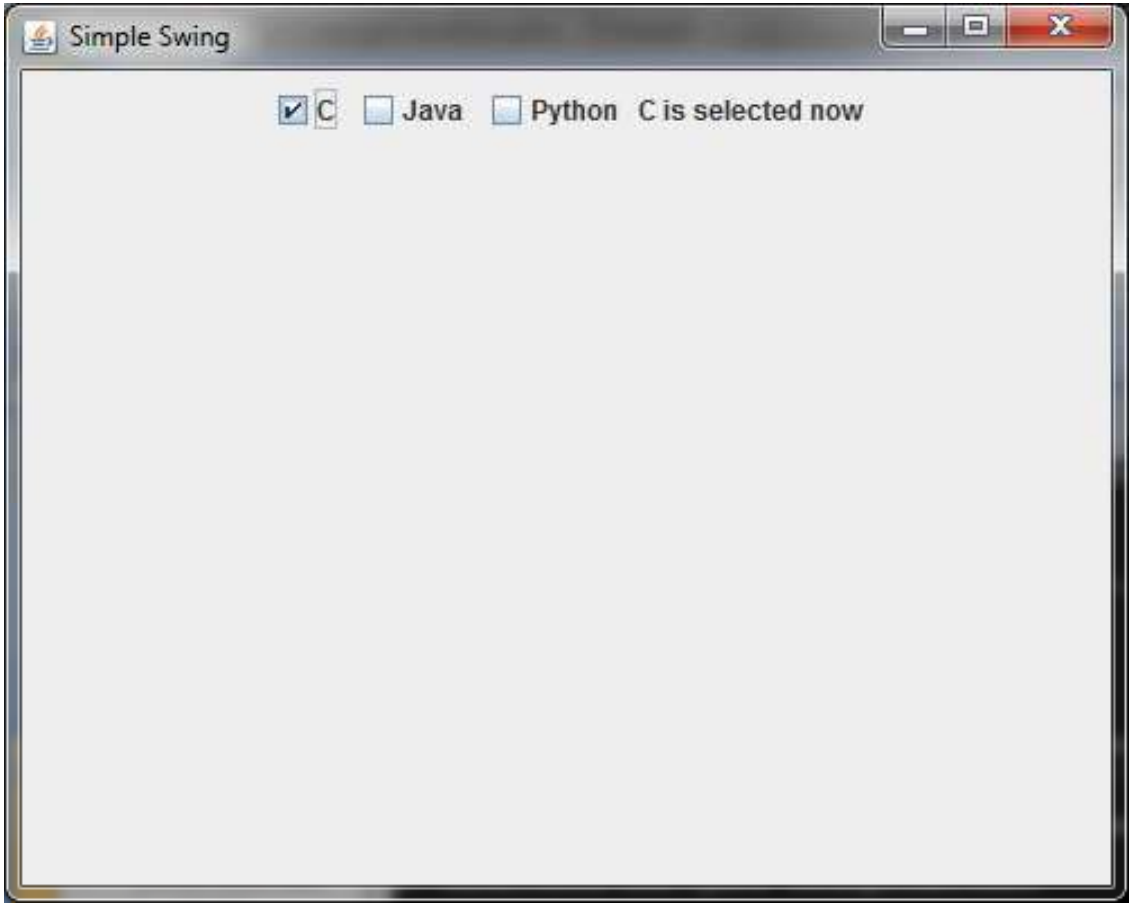
JCheckBox

- **JCheckBox** class provides the functionality of a check box.
- Its immediate **superclass** is **JToggleButton**,
- **JCheckBox** defines several constructors.

  JCheckBox(String *str)*
- When the user selects or deselects a check box, an ItemEvent is generated.
- Inside the **itemStateChanged**( ) method, **getItem**( ) is called on ItemEvent object to obtain a reference to the JCheckBox object that generated the event
- To determine the selected state of a check box is to call **isSelected**( ) on the **JCheckBox** instance.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class SwingJCheckBox extends JFrame
    implements ItemListener
{   JFrame jfrm;
     JCheckBox cb;
    JLabel jlab;
SwingJCheckBox()
    {
jfrm = new JFrame("Simple Swing ");
jfrm.setSize(500, 400);
jfrm.setLayout(new FlowLayout());
jfrm.setDefaultCloseOperation(JFrame
    .EXIT_ON_CLOSE);
 JCheckBox cb = new JCheckBox("C");
cb.addItemListener(this);
jfrm.add(cb);
cb = new JCheckBox("Java");
cb.addItemListener(this);
jfrm.add(cb);
cb = new JCheckBox("Python");
cb.addItemListener(this);
jfrm.add(cb);

jlab = new JLabel("Select language");
jfrm.setVisible(true);
 }
```

```
public void itemStateChanged(ItemEvent ie)
    {       JCheckBox cb = (JCheckBox)ie.getItem();
            if(cb.isSelected())
                    jlab.setText(cb.getText() + " is selected now");
            else
                    jlab.setText(cb.getText() + " is cleared now");
    }
public static void main(String args[])
    {               SwingUtilities.invokeLater(new Runnable()
                    {
                            public void run()
                            {
                                    new SwingJCheckbox();
                            }
                    });
    } }
```


Simple Swing
C
Java
Python
C is selected now

JRadioButton

- Radio buttons are a group of **mutually exclusive buttons**, in which **only one button can be selected at any one time.**
- They are supported by the **JRadioButton class**, which extends JToggleButton.
- **JRadioButton** provides several constructors

JRadioButton(String *str)*

- A button group is created by the **ButtonGroup** class.
- Its default constructor is invoked for this purpose.
- Elements are then added to the button group via the following method:

void **add**(AbstractButton *ab)*

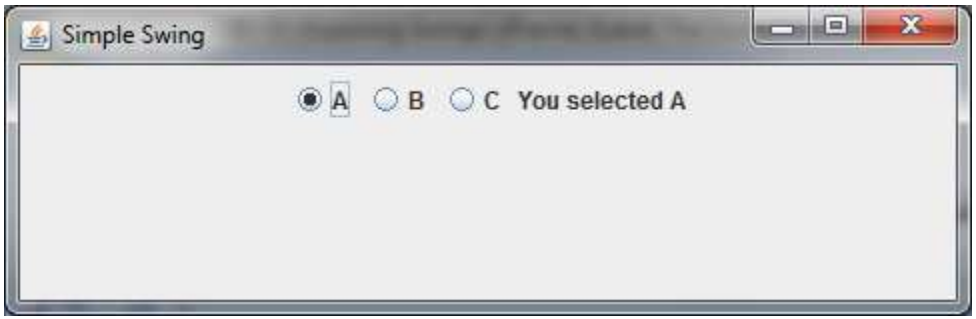
  - Here, *ab is a reference to the button to be added to the group.*
- A **JRadioButton** generates **action events, item events,** and **change**

JRadioButton(contd.)

- We will normally implement the **ActionListener interface** with method **actionPerformed( ).**
  - Inside this method we can check its action command by calling **getActionCommand( ).**
    - By **default**, the action command is the same as the **button label**, but we can set the action command to something else by calling **setActionCommand( ) on the radio button.**
- We can call **getSource( ) on the ActionEvent object and check that reference against the buttons.**
- We can simply check each radio button to find out which one is currently selected by calling **isSelected( )** on each button.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class SwingJRadioButton extends
    JFrame implements ActionListener
{   JFrame jfrm;
    JLabel jlab;
SwingJRadioButton()
    {
jfrm = new JFrame("Simple Swing ");
jfrm.setSize(220, 100);
jfrm.setLayout(new FlowLayout());
jfrm.setDefaultCloseOperation(JFrame
    .EXIT_ON_CLOSE);
JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        jfrm.add(b1);
JRadioButton b2 = new JRadioButton("B");
        b2.addActionListener(this);
        jfrm.add(b2);
JRadioButton b3 = new JRadioButton("C");
        b3.addActionListener(this);
        jfrm.add(b3);
    ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);
    jlab = new JLabel("Select language");
    jfrm.setVisible(true);
    }
```

```
public void actionPerformed(ActionEvent ae)
    {
        jlab.setText("You selected " + ae.getActionCommand());
    }
    public static void main(String args[])
    {           SwingUtilities.invokeLater(new Runnable()
                {
                    public void run()
                    {
                        new SwingJRadioButton ();
                    }
                });
    } }
```



JTextField.

- **JTextField** is the simplest Swing text component.
- JTextField allows you to **edit <u>one line</u> of text.**
  - It is derived from **JTextComponent,** which provides the basic functionality common to Swing text components.
- Three of **JTextField's constructors** are **:**

  JTextField(int *cols)*

  JTextField(String *str, int cols)*

  JTextField(String *str)*
  - Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty.
  - If the number of columns is not specified, the text field is sized to fit the specified string

JTextField(contd.)

- **JTextField** generates events in response to user interaction.
  - For example, an **ActionEvent** is fired when the user presses ENTER.
  - A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position.
    - **CaretEvent** is packaged in **javax.swing.event**
- To obtain the text currently in the text field, call **getText( )**

JTextField(contd.)


```
// A simple Swing application.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/* <object code="SwingText" width=220 height=90>
</object>
*/
public class SwingText extends JApplet implements ActionListener
{   JLabel jlab;
    JTextField jtf;
```

```
private void makeGUI()
    {     setLayout(new FlowLayout());
          jlab = new JLabel(" Swing is powerful GUI");
          add(jlab);
          jtf = new JTextField(15);
          jtf.addActionListener(this);
          add(jtf);
    }
public void actionPerformed(ActionEvent ae)
    {
          showStatus(jtf.getText());
    }
```

```
public void init()
    {     try {     SwingUtilities.invokeAndWait(new Runnable ()
                                        {
                                        public void run()
                                        {
                                                makeGUI();              }
                                        });
                        } catch(Exception exc)
            { System.out.println("Can't create because of "+ exc); }
    }
}
```

COMPILE USING
javac SwingText.java
RUN
appletviewer SwingText.java

JList

- In Swing, the basic list class is called **JList.**
- **JList** provides several constructors

  JList(Object[ ] *items)*

- A JList generates a **ListSelectionEvent** when the user makes or changes a selection or deselects an item. It is handled by implementing **ListSelectionListener**
- **ListSelectionListener** interface specifies only one method, called **valueChanged(),**

  void **valueChanged**(ListSelectionEvent *le)*

JList(contd.)

- We can change this behavior by calling **setSelectionMode( ),**

  void **setSelectionMode**(int *mode)*

  Here *mode can be*

  SINGLE_SELECTION

  SINGLE_INTERVAL_SELECTION

  MULTIPLE_INTERVAL_SELECTION

- We can obtain the index of the item selected from list by calling **getSelectedIndex( ) :**

  int **getSelectedIndex**( )

  – Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned.

- We can obtain the value associated with
- the selection by calling **getSelectedValue( ):**

  Object **getSelectedValue**( )

JComboBox

- Swing provides a combo box (a combination of a text field and a drop-down list) through the **JComboBox class.**

- **JComboBox constructor** is:

  JComboBox(Object[ ] *items)*

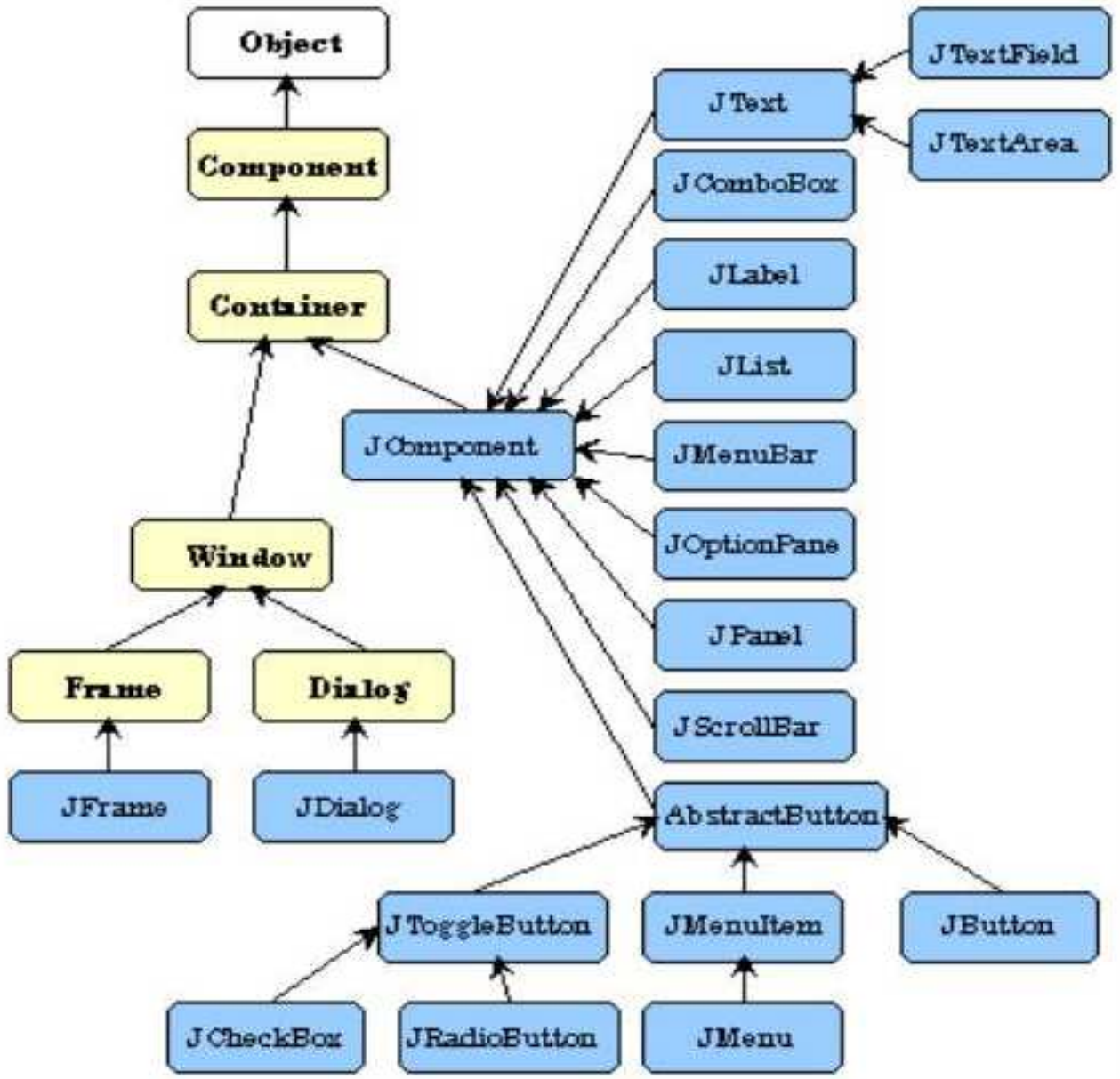
- Items can also be dynamically added to the list of choices via the **addItem( ) method:**

  void **addItem**(Object *obj)*

- *To* obtain the item selected in the list is to call **getSelectedItem( ) on the combo** box.

Object **getSelectedItem**( )

Class hierarchy of swing components



Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 5 - **Graphical User Interface and Database support of Java**

(Part 6)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **JDBC overview**
  - ☑ Creating and Executing Queries
    - ☑ create table
    - ☑ delete
    - ☑Insert
    - ☑select

Introduction

- Programming Language -**Java**
  - for coding, developing interfaces(**front end** of an application)
- Database – **MySQL , Oracle** , **PostgreSQL**
  - For storing data- (**back end** of an application)
    - Different types- relational database, object based database etc.
  - Relational database
    - Tabular structure
    - **E.g.** - Oracle, Microsoft Access, MySQL, PostgreSQL, MongoDB etc
- SQL- Structured Query Language
  - **SQL** statements are used to perform operations on a database. We can insert , delete, update and retrieve data in database using SQL statements.

JDBC overview

- JDBC stands for "**Java DataBase Connectivity**".
- JDBC API (***Application Programming Interface***) is a Java API that can access any kind of **tabular data**, especially data stored in a ***Relational database*.**
- JDBC is used for executing SQL statements from Java program.
- With the help of JDBC API, we can insert, update, delete and fetch data from the database.

Why JDBC?

- Before JDBC, **ODBC**(**O**pen **D**ata**B**ase **C**onnectivity) API was the database API to connect and execute the query with the database.
  - But, ODBC API uses ODBC driver which is written in C language (i.e. *platform dependent and unsecure*).
  - That is why, Java has defined its own API (**JDBC API**) that uses JDBC drivers (*written in Java language*).
- If our application is using JDBC API to interact with the database, then we need not change much in our code even if we change the database of our application.

Advantage of JDBC

- JDBC **standardizes** how to do many of the operations like
  - connect to the database,
  - query the database,
  - update the database, and
  - call stored procedures.

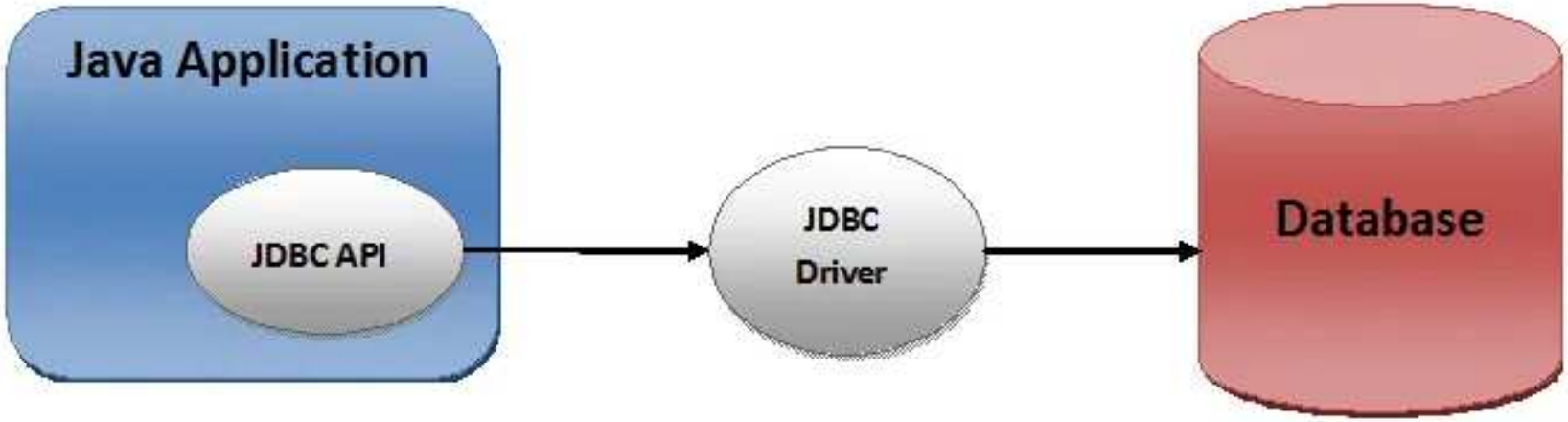
JDBC architecture

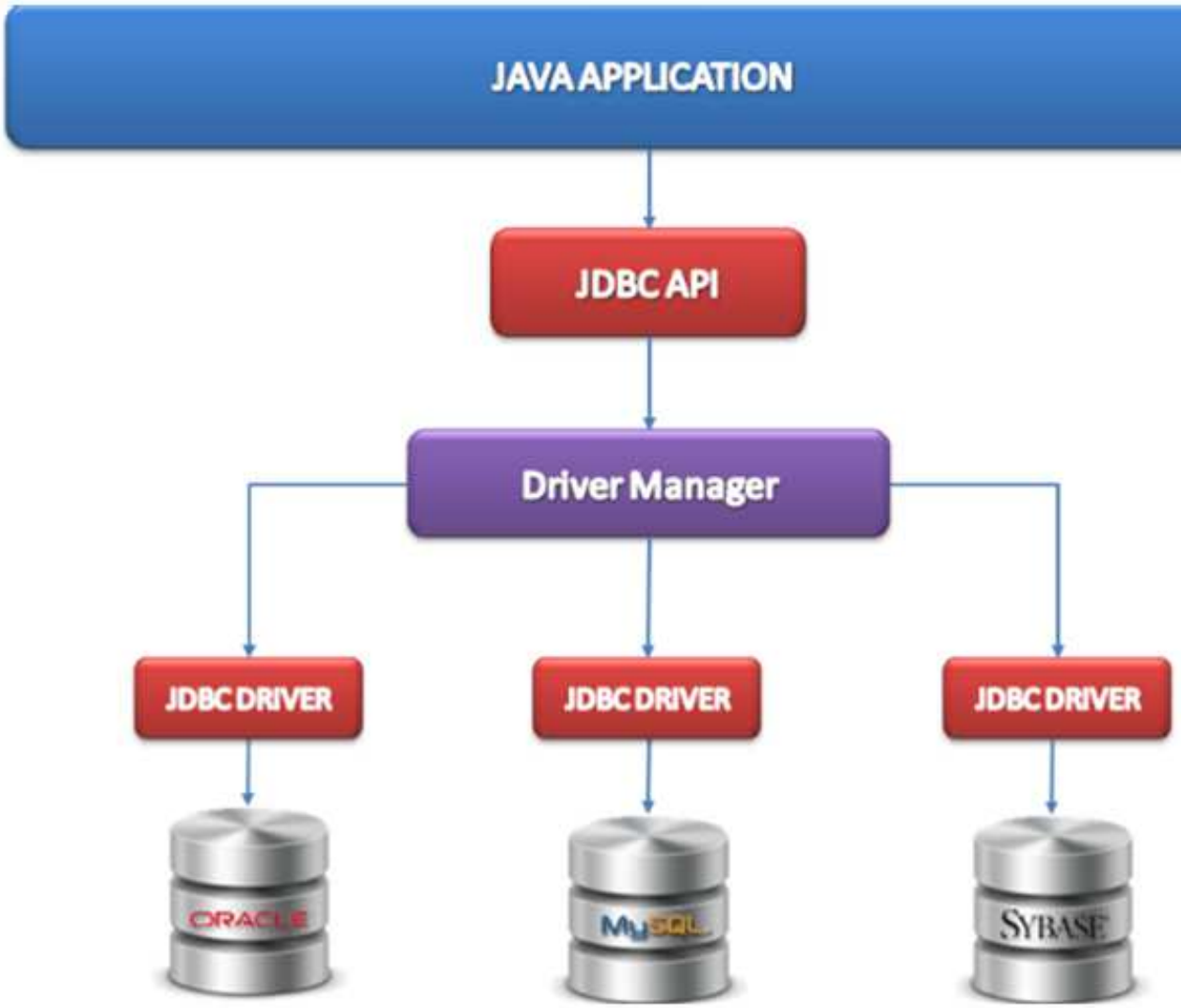
- **JDBC** architecture can be classified in 2 broad categories:-

1. JDBC API
2. JDBC Drivers

Java-JDBC API -JDBC Driver- Database

- **Java** – programming language – coding- Front end
- The **JDBC API** *defines a set of interfaces and classes that all major database providers* follow, so that using JDBC API, Java developers can connect to many Relational Database Management Systems (RDBMS).
  - **JDBC API** uses **JDBC drivers** to connect with the database.
- A **JDBC driver** is a software component that enables a Java application to interact with specific database.
- **Database** – store data – Back end


Java Application
JDBC API
JDBC
Driver
Database


JAVA APPLICATION
JDBC API
Driver Manager
JDBC DRIVER
JDBC DRIVER
JDBC DRIVER
ORACLE
MySQL
SYBASE

JDBC API

- The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language
- Using the **JDBC API**, we can access virtually any data source like relational databases , spreadsheets etc. from Java.
- The JDBC API is comprised of two packages:

  java.sql

  javax.sql

- These packages contains classes and interfaces for JDBC API.

JDBC API

- Some classes and interfaces in **java.sql** package which support connectivity between Java and database are:
  - **DriverManage**r :-"DriverManager class" **manages** all the Drivers found in JDBC environment, *load the most appropriate driver* for connectivity.
  - **Connection**: Connection interface objects which represents connection and it's object also helps in *creating object of Statement*, PreparedStatement etc.
  - **Statement** : :-Statement interface object is used to *execute query* and also store it's value to "ResultSet" object.

JDBC API(contd.)

- **PreparedStatement**:- represents a *precompiled SQL statement* .
- **Callable Statement**:-Callable statement *support stored procedure* .
- **ResultSet**: it is used to store the result of SQL query. Java application get the result of database from this ResultSet.
- **SQLException**:- SQLException class is used to represent *error or warning* during access from database or during connectivity.

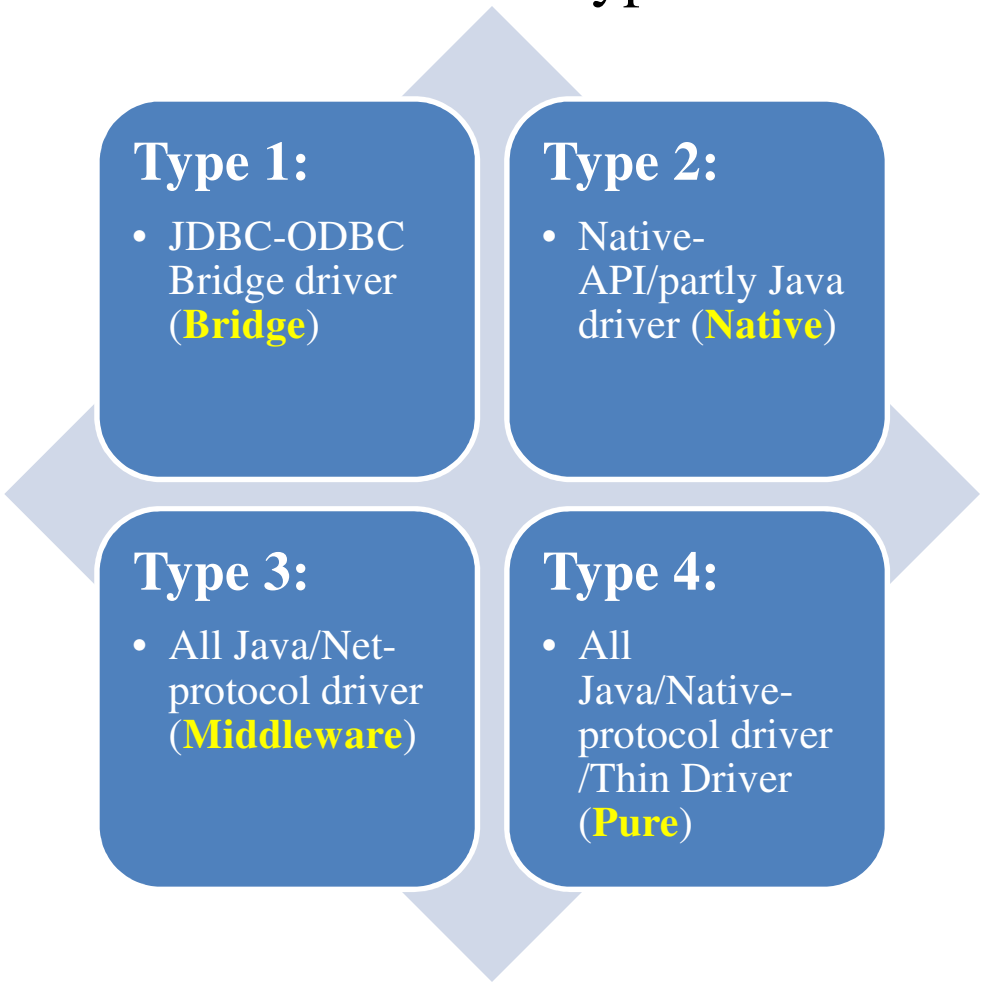
JDBC Drivers

- **JDBC API** uses **JDBC drivers** to connect with the database.
- JDBC drivers .All major vendors provide their own JDBC drivers .
- A **JDBC driver** is a software component that enables a Java application to interact with a database.
- JDBC drivers contain a set of java classes that enables to connect to that particular database.

JDBC Drivers

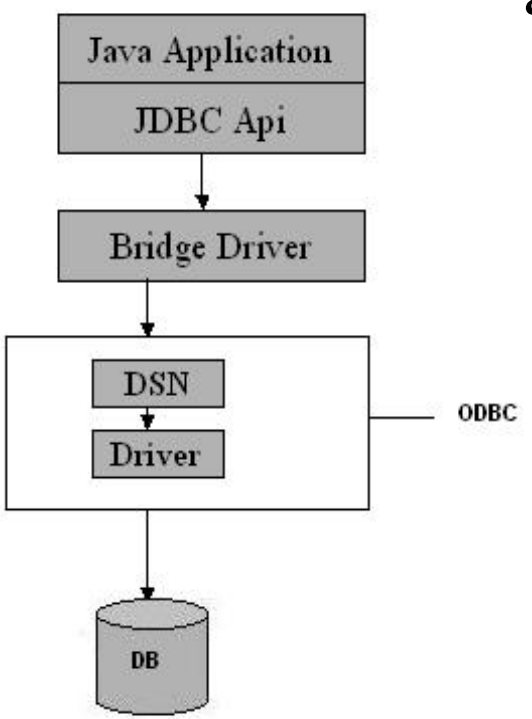
Types Of JDBC Drivers:

JDBC drivers are divided into four types or levels.



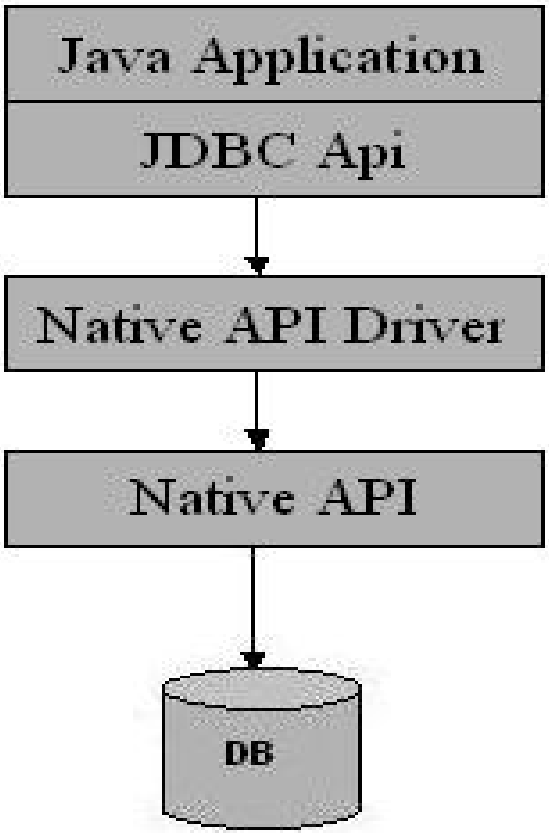
Type 1 JDBC Driver

- The Type 1(**JDBC-ODBC Bridge driver)** driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver
- **Advantage**
  - The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.



Type 2 JDBC Driver

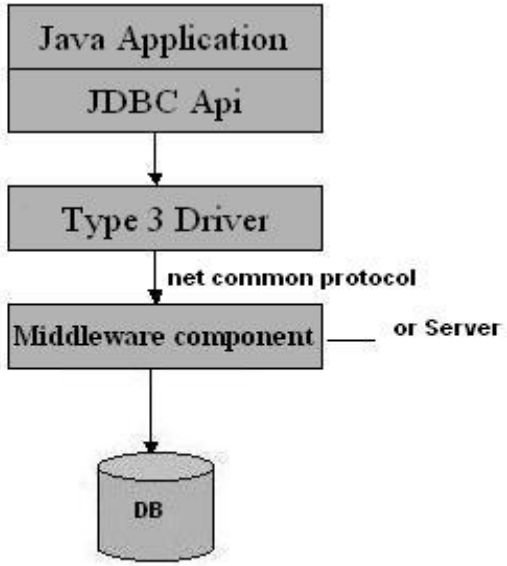
- Type 2 drivers(**Native-API/partly Java driver**) convert JDBC calls into database-specific calls.



Type 3 JDBC Driver

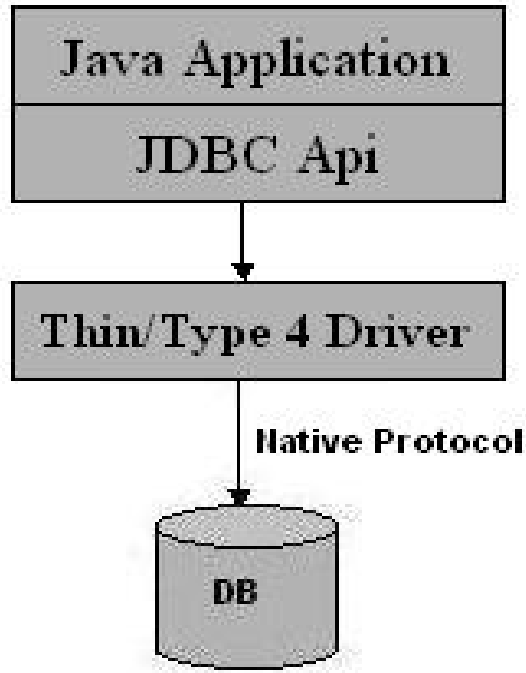
- Type 3 database driver(**All Java/Net-protocol driver**) requests are passed through the network to the middle-tier server.
- The middle-tier then translates the request to the database.

Advantage
This driver is *fully written in Java* and hence *portable*.
It is *suitable for the web*.



Type 4 JDBC Driver

- Type 4 drivers(**Native-protocol/all-Java driver)** uses java networking libraries to communicate directly with the database server.
- Advantage
  - They are completely written in Java-platform independent.
  - It is most suitable for the web.

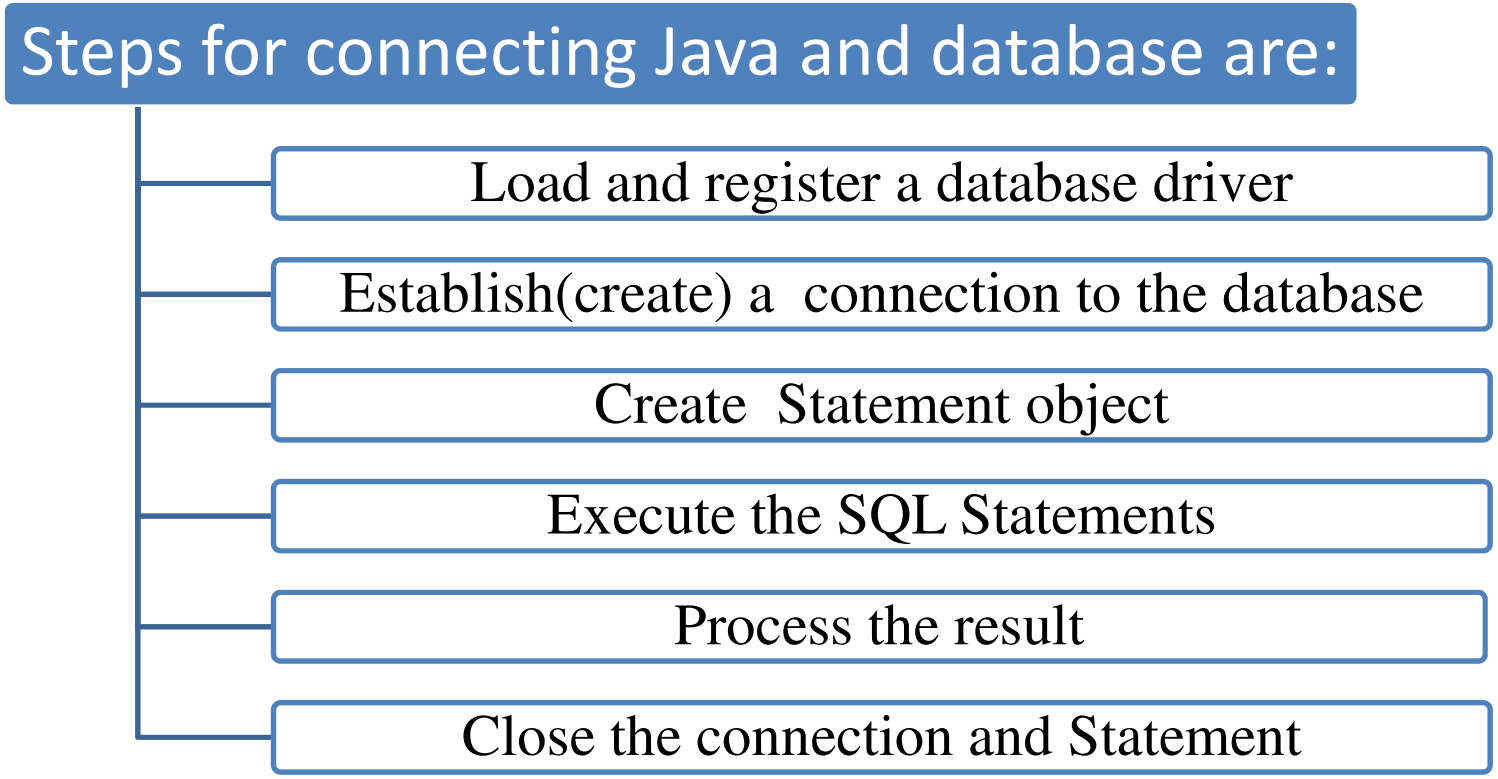


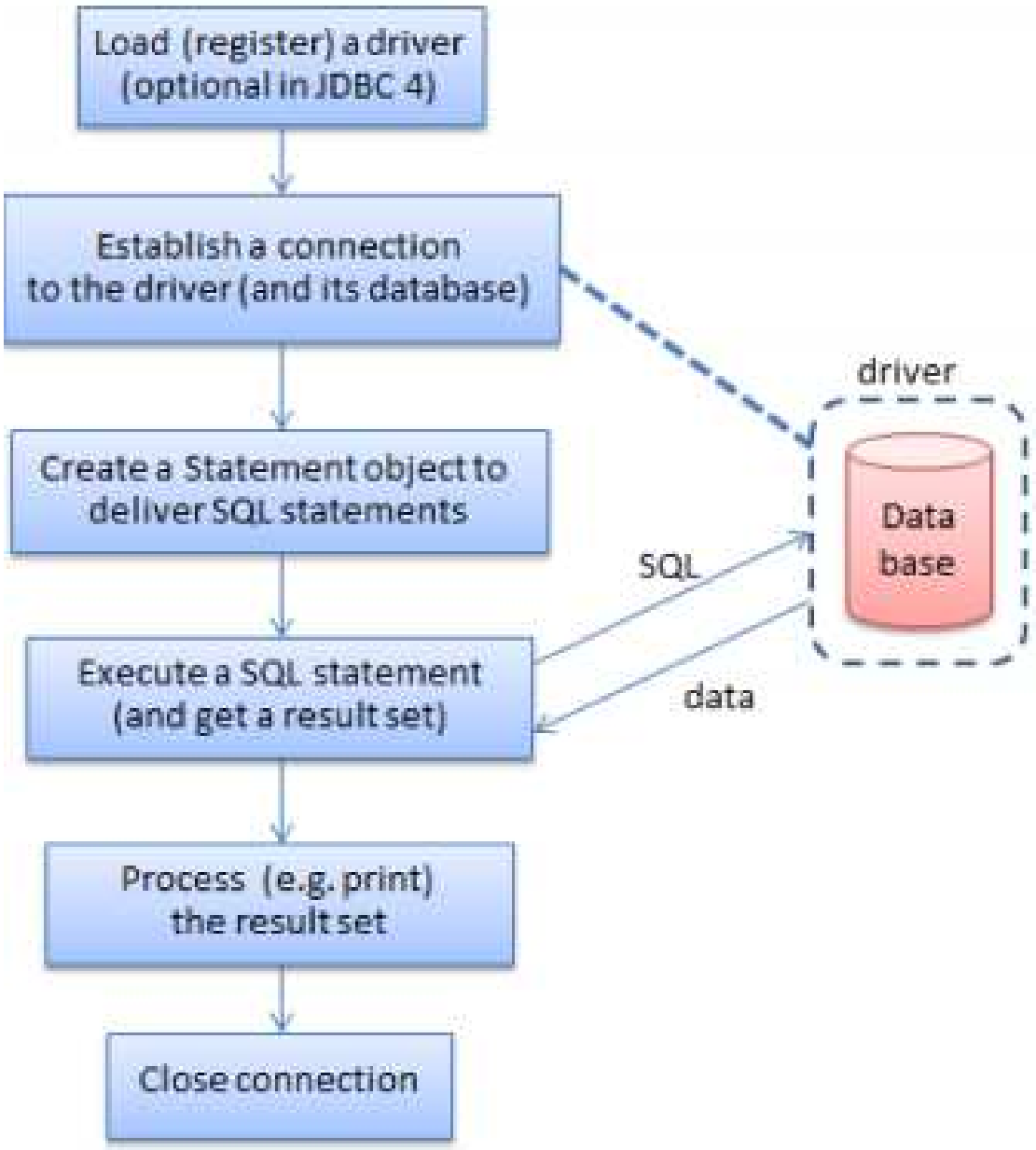
Java-Database connectivity

Basic Steps fo connecting Java and database

- Before we can create a java JDBC connection to the database, we must first import the **java.sql package** using:-

import java.sql.*;




Load (register) a driver
(optional in JDBC 4)
Establish a connection
to the driver (and its database)
Create a Statement object to
deliver SQL statements
Execute a SQL statement
(and get a result set)
Process (e.g. print)
the result set
Close connection
driver
Data
base
SQL
data

- **Steps to develop JDBC application:**
  1. Load and Register Driver
  2. Establish the connection between java application and database
  3. Creation of statement object
  4. Send and execute SQL query
  5. Process Result from ResultSet
  6. Close the connection and statement.

Java Database Connectivity steps

1. Load or register a database driver

- We can load /register a driver in Java in one of two ways:
  - **Class.forName(String *driver*)**
  - **DriverManager.registerDriver(new *constructor of the driver*)**
- We can load the driver class by calling **Class.forName()** with the Driver class name as an *argument* or **DriverManager.registerDriver()** with *constructor of the driver class as argument*
  - Once loaded, the Driver class creates an instance of itself.
  - JDBC-ODBC Bridge driver is commonly used.
- Each database has its own driver.
- The JDBC Driver class for MySQL database are

  com.mysql.jdbc.Driver

  com.mysql.cj.jdbc.Driver

Java Database Connectivity

- The code for **loading MySQL database driver from Java**

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

or

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
```

```
Class.forName("com.mysql.jdbc.Driver");
```

Or

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

Java Database Connectivity(contd.)

2. Establish the connection between Java application and database

– The **getConnection**() method of **DriverManager** class is used to establish connection with the database.

public static Connection getConnection(String ***url*** ,String ***name***, String ***password***) **throws SQLException**

```
Connection con=DriverManager.getConnection ( "jdbc:mysql://localhost:3306/ABC" , "root" , "root123" );
```

JDBC url

username

password

Java Database Connectivity(contd.)

- A JDBC URL provides a way of identifying a database so that the appropriate driver will recognize it and establish a connection with it.

The standard syntax for JDBC URLs is:

jdbc:<***subprotocol***>:<***subname***>

- A JDBC URL has three parts, which are separated by colons:
  - **jdbc** is the protocol.
  - <**subprotocol**> is usually the **driver** or the database connectivity mechanism, which may be supported by one or more drivers.
  - <**subname**> is the database.
  - For example, to access a MySQL database through a JDBC-ODBC bridge, one might use a URL such as the following:

jdbc:mysql://localhost:3306/ABC

Java Database Connectivity(contd.)


```
Connection con=DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/databasename" , "username",
   "password"   );
```

- The JDBC url used is

jdbc:mysql://localhost:3306/databasename

  - Here **jdbc** is the API,
  - **mysql** is the database,
  - **localhost** is the server name on which **mysql** is running(we can give IP address here)
  - **3306** is the port number
  - **databasename** is the name of the database created in MySQL

- **username** Give the username of MySQL(root is the default username. Other MySQL username also we can give here).
- **password** Give the password of MySQL login given during installation.

Java Database Connectivity(contd.)

3. Creating a Statement object

- Once a connection is established, we can interact with the database.
- To execute SQL statements, we need to create a Statement object from the Connection object by using the createStatement() method.
- A Statement object is used to send and execute SQL statements to a database.

Statement object= connectionobject. **createStatement();**

E.g.

```
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/ABC" , "root" , "root123" );
```

Statement st = con.createStatement();

- Statement object **st** is used to **send and execute SQL statements** to a database.

Java Database Connectivity(contd.)

4. Execute the SQL Statements

- To execute a SQL statement we use **executeQuery**() or **executeUpdate() method** on **Statement** object**.**
  - ❑ The **executeUpdate()** method executes the **CREATE, INSERT, DELETE** and **UPDATE** statements.
    - E.g.:

    st.executeUpdate("CREATE TABLE stud1(roll int, name varchar(15))");
    st.executeUpdate("INSERT INTO student values(1, 'Anu') ");

  - ❑ The ***executeQuery***() method executes a **SELECT** query
    - it takes an SQL SELECT query string as an argument and returns the result(output) as a ***ResultSet*** object.
    - *E.g.:*

    ResultSet rs=st.***executeQuery***("SELECT rollno, name from student");

Java Database Connectivity(contd.)

5. Process the Result (in the case of SELECT query only -To Retrieve the Result)

- The result of SELECT query (retrieves data from database) is stored in **ResultSet** object.
- To retrieve the data from the ResultSet object, we have to use ResultSet's getxxx() method (where xxx is the data type.)

  public int getxxx(int columnIndex):

  public int getxxx(Stringt columnIndex):

  - getInt() to retrieve a integer value.
  - getString() method can be used to retrieve a string value.
  - getFloat() method can be used to retrieve a floating point value.

Java Database Connectivity(contd.)

Eg:

```
ResultSet  rs=st.executeQuery("SELECT  rollno,  name
                    from  student");
while(rs.next())
{
        System.out.println(rs.getInt(1));
        System.out.println(rs.getString(2));
}
```

- Here rs contains all rollno and name rows in studenttable
- **ResultSet cursor** is initially positioned **before the first row;**
  - **the first call to the method next()** (rs.next()) moves the cursor forward and makes the first row the current row)
  - the second call to next() moves the cursor forward and makes the second row as the current row, and so on.

Java Database Connectivity(contd.)

- rs.getInt(1) will retrieve the value of first attribute in the current row(getInt() is used because first attribute is rollno which is integer)
- rs.getString(2) will retrieve the value of second attribute in the current row(getString() is used because second attribute is name which is character string)

Java Database Connectivity(contd.)


```
ResultSet rs=st.executeQuery("SELECT rollno, name from   student");
        while(rs.next())
        {
                System.out.println(rs.getInt(1));
        System.out.println(rs.getString(2));
        }
```

Working:

- Here rollno and name in all rows in student table are retrieved.
- **ResultSet** object **rs** maintains a cursor that is initially positioned **before the first row.**
- When while(rs.**next()**) is first executed the cursor moves forward to first row of result and prints the value of first attribute (rollno) and second attribute(name)in first row in the result
- Next time while loop is executed, if second row is there, then rs.nex() moves cursor to second row and prints the values in thar row.
- This continues until there is no more row in the result.

Java Database Connectivity(contd.)

6. Close the connection and Statement

- Finally open connections need to be closed using **close()** method as:

con.**close**()

st. **close**()

MySQL

- MySQL is an open source database software.
  - We can create databases, tables etc for storing data and we can perform various database operations.
- Take mysqlshell and type:

\sql

\connect root@localhost

Enter password **root123** **(password given during installation)**

- Create database

CREATE DATABASE ABC;

- Use the database for creating tables

USE ABC;

- Now you can do database operations in MySQL

SQL Commands- CREATE TABLE

- **CREATE TABLE** tablename (atribute1 type, attribute2 type, attribute3 type Primary Key,….. atributen type,***constraints***);
  - Type can be
    - int
    - char(size)
    - varchar(size)
    - real
    - date
- E.g. CREATE TABLE person(name varchar(15) , age int);

SQL Commands- INSERT

- Insert rows(values of attribute) into table.
- **INSERT INTO** tablename (attribute,attribute..) **VALUES**(value1,value2,….);
- If type of attribute is varchar or char its value should be enclose in single quotes.
- E.g. Insert the following details into Person table
- Name is Anu Age 20, Name is Smith Age 10 ,

  Name is Roy Age 70

INSERT INTO Person(name,age) VALUES('Anu', 20);

INSERT INTO Person(name,age) VALUES('Smith', 10);

INSERT INTO Person(name,age) VALUES('Roy', 70);

SQL Commands- DELETE

- Delete from table.

DELETE FROM tablename **WHERE** condition;

- Condition can be of the form:

  Attribute=value Attribute<value Attribute>value

  Attribute<=value Attribute>=value Attribute!=value

  Attribute BETWEEN value 1AND value2

  – If more than one condition is there, then they can be combined using AND , OR as required.

E.g. . *Remove details of persons having age 20*

DELETE FROM Person WHERE age=20;

Remove details of persons having name Smith or age more than 60

DELETE FROM Person where age>60 OR name='Smith';

SQL Commands - UPDATE

- To update or modify the contents in table.

  UPDATE tablename **SET** attribute=value , attribute=value **WHERE** condition;

- Condition can be of the form:

  Attribute**=**value Attribute**<**value Attribute**>**value

  Attribute**<=**value Attribute**>=**value Attribute**!=**value

  Attribute BETWEEN value 1AND value2

  – If more than one condition is there, then they can be combined using AND , OR as required.

E.g. *Change the age of Roy to 25*

UPDATE Person SET age=25 WHERE name='Roy';

SQL Commands - SELECT

- To select or retrieve content from table.

  SELECT attribute1, attribute2,….. attributen **FROM** tablename **WHERE** condition;

- Condition can be of the form:

  Attribute**=**value Attribute**<**value Attribute**>**value

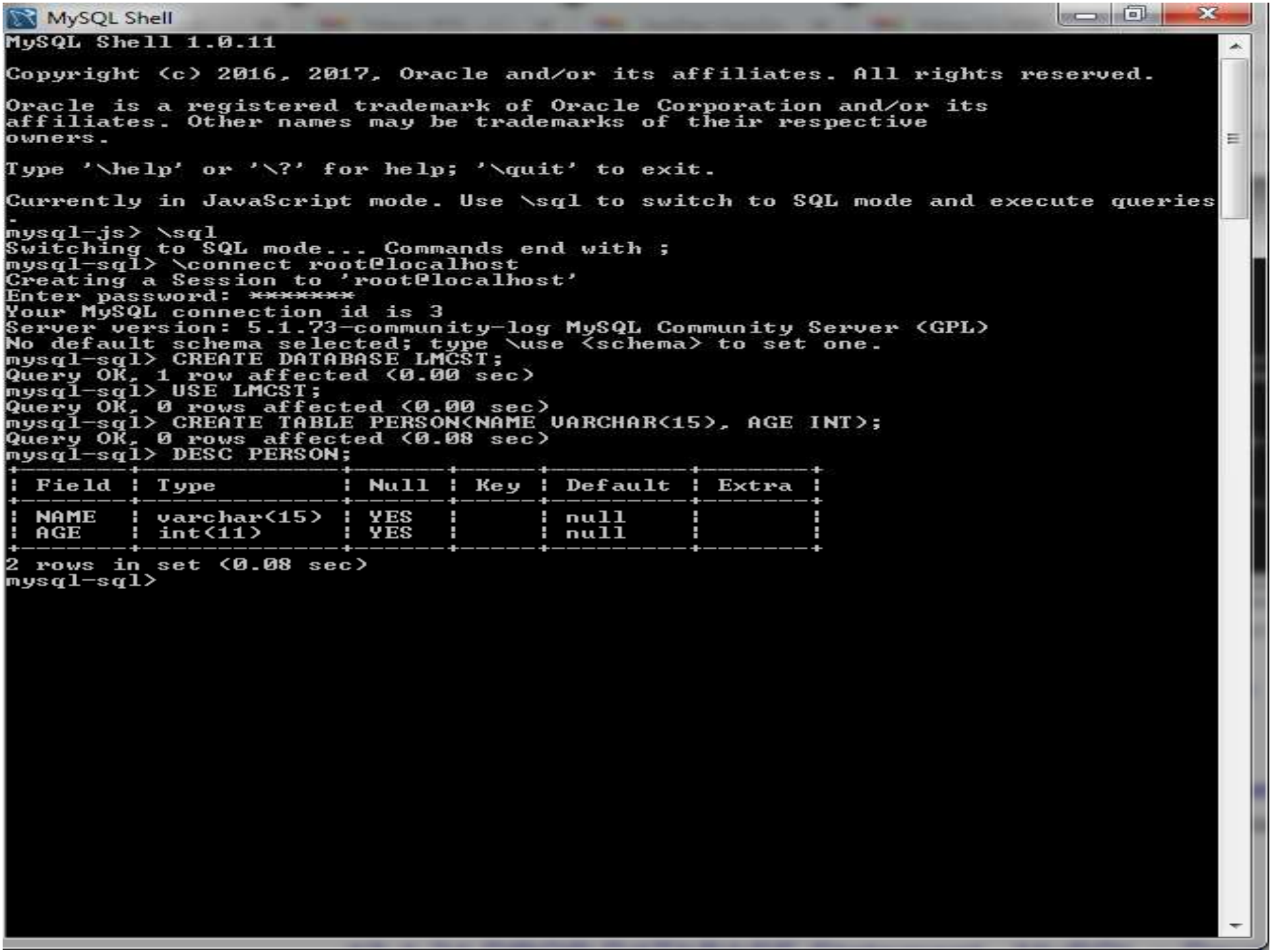
  Attribute**<=**value Attribute**>=**value Attribute**!=**value

  Attribute BETWEEN value 1AND value2

  – If more than one condition is there, then they can be combined using AND , OR as required.

E.g. *Display name and age of persons with age more than 10.*

SELECT name, age FROM Person WHERE age>10;


MySQL Shell
MySQL Shell 1.0.11
Copyright (c) 2016, 2017, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type '\help' or '\?' for help; '\quit' to exit.
Currently in JavaScript mode. Use \sql to switch to SQL mode and execute queries
.
mysql-js> \sql
Switching to SQL mode... Commands end with ;
mysql-sql> \connect root@localhost
Creating a Session to 'root@localhost'
Enter password: *******
Your MySQL connection id is 3
Server version: 5.1.73-community-log MySQL Community Server (GPL)
No default schema selected; type \use <schema> to set one.
mysql-sql> CREATE DATABASE LMCST;
Query OK, 1 row affected (0.00 sec)
mysql-sql> USE LMCST;
Query OK, 0 rows affected (0.00 sec)
mysql-sql> CREATE TABLE PERSON(NAME VARCHAR(15), AGE INT);
Query OK, 0 rows affected (0.08 sec)
mysql-sql> DESC PERSON;
+-------+-------------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| NAME | varchar(15) | YES | | null | |
| AGE | int(11) | YES | | null | |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.08 sec)
mysql-sql>

- Creating and Executing Queries

✓ **CREATE TABLE**

✓ **DELETE**

✓**INSERT**

✓**SELECT.**

CREATE TABLE in MySQL from Java

- Write a Java program to create a table named **Student** with fields **rollno** and **name** in the database ABC

Attribute	Domain type
rollno	int
name	Varchar(15)

Create table from Java


```
import java.sql.*;
public class CreateTableEg
{   public static void main(String args[]) throws ClassNotFoundException
    {
    try {
 // Register JDBC driver
     Class.forName("com.mysql.jdbc.Driver");
//Open a connection
    Connection con=DriverManager.getConnection(
          "jdbc:mysql://localhost:3306/ABC" , "root","root123");
 //Create a statement object
    Statement st=con.createStatement();
    st.executeUpdate("CREATE TABLE Student (rollno int, name varchar(15))");
    System.out.println("Table created");
    con.close();                              //Close the connection
          }catch(SQLException e) {      System.out.println("Error is " +e);    }

    }
}
```

INSERT row using Java

- Write a Java program to insert the following row into **Student** table with fields **rollno** and **name** in the database ABC

1	**Anu**

INSERT row using Java


```
import java.sql.*;
public class InsertEg
{   public static void main(String args[]) throws ClassNotFoundException
    {
    try {
 // Register JDBC driver
     Class.forName("com.mysql.jdbc.Driver");
//Open a connection
    Connection con=DriverManager.getConnection(
          "jdbc:mysql://localhost:3306/ABC" , "root","root123");
 //Create a statement object
    Statement st=con.createStatement();
     st.executeUpdate("INSERT INTO Student(rollno,name) VALUES (1,'Anu')");
    System.out.println("Data inserted successfully");
    con.close();                                  //Close the connection
          }catch(SQLException e) {       System.out.println("Error is " +e);    }
    }
}
```

- To access value of java **variable** inside the SQL command use :
  *Singlequote*Doublequote+**varaiable+**Doublequote*Singlequote*
- E.g.

```
int roll=2;
String nam="Anu";
st.executeUpdate("INSERT INTO Student(rollno,name)
                        VALUES('"+ roll +"'  , '"+ nam +"'  )");
```

INSERT row using Java

- Write a Java program to insert details about n students into **Student** table with fields **rollno** and **name** in the database ABC

INSERT n rows from Java


```
import java.sql.*;     import java.util.*;
public class CreateTableEg
{   public static void main(String args[]) throws ClassNotFoundException
    { try {          Scanner sc=new Scanner(System.in);
                     int roll,n,i;
                     String nam;
     Class.forName("com.mysql.jdbc.Driver");
    Connection con=DriverManager.getConnection(
          "jdbc:mysql://localhost:3306/ABC" , "root","root123");
    Statement st=con.createStatement();
     System.out.println("Enter how many students detail to be inserted");
    n=sc.nextInt();
    for(i=0;i<n;i++)
    {      System.out.println("Enter the rollno");
           roll=sc.nextInt();
           System.out.println("Enter the name");
           sc.nextLine();
           nam=sc.nextLine();
st.executeUpdate("INSERT INTO Student(rollno,name)
                     values('"+ roll +"'  , '"+ nam +"' )");
    }      System.out.println("Data inserted successfully");
     con.close();
    }
catch(SQLException e)
      {
       System.out.println(e);
      }
    }
}
```

DELETE rows using Java

- Write a Java program to delete details about students into **Student** table with given roll number.

DELETE rows using Java


```
import java.sql.*;     import java.util.*;
public class InsertEg
{   public static void main(String args[]) throws ClassNotFoundException
    {       int roll;
            Scanner sc=new Scanner(System.in);
    try {   Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/ABC" , "root","root123");
            Statement st=con.createStatement();

            System.out.println("Enter the rollno of student to be deleted");
            roll=sc.nextInt();
    st.executeUpdate("DELETE FROM Student WHERE rollno='"+ roll +"' ");
            System.out.println("Data deleted succesfully.");
            con.close();
            }catch(SQLException e) {       System.out.println("Error is " +e);    }
    }
}
```

UPDATE rows using Java

- Write a Java program to update the name of roll number 1 to John in **Student** table

UPDATE rows using Java

```
import java.sql.*;    import java.util.*;
public classUpdateEg
{    public static void main(String args[]) throws ClassNotFoundException
     {        Scanner sc=new Scanner(System.in);
              int roll;                String nam;
              try {       Class.forName("com.mysql.jdbc.Driver");
              Connection con=DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/ABC" , "root","root123");
              Statement st=con.createStatement();
               System.out.println("Enter the rollno of student ");
              roll=sc.nextInt();                              sc.nextLine();
              System.out.println("Enter the name of new student ");
              nam=sc.nextLine();
st.executeUpdate(''UPDATE  Student SET name=''+ nam +'''  WHERE
                                  rollno=''+ roll +''' ');
              con.close();
              }catch(SQLException e) {       System.out.println("Error is " +e);    }
     }
}
```

SELECT rows using Java

- Write a Java program to **list all names and roll numbers** in **Student** table

SELECT rows Using Java


```
import java.sql.*;
public class InsertEg
{   public static void main(String args[]) throws ClassNotFoundException
    {
    try {
          Class.forName("com.mysql.jdbc.Driver");
          Connection con=DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/ABC" , "root","root123");
          Statement st=con.createStatement();
          ResultSet rs=st.executeQuery("SELECT rollno,name FROM student");
          while(rs.next())
          {                    System.out.println(rs.getInt(1)+"  "+rs.getString(2));
          }
          con.close();
      }catch(SQLException e) {      System.out.println("Error is " +e);    }
    }
}
```

Import classes and interfaces from java.sql package

Load the Driver for MySQL

Establish connection

Create statement

Execute Query and represent result as result set

First column

Second column

Close the connection

Load the Driver for MySQL

SELECT rowsUsing Java

```
import java.sql.*;
public class InsertEg
{   public static void main(String args[]) throws ClassNotFoundException
    {
    try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/ABC" , "root","root123");
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery("select rollno,name from student");
            while(rs.next())
            {                   System.out.println(rs.getInt(1)+"  "+rs.getString(2));
            }
            con.close();
       }catch(SQLException e) {      System.out.println("Error is " +e);    }
    }
}
```

WORKING

- In this example we used MySQL as the database.
- **Class**.forName() is used for loading the Driver class
- **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
- DriverManager.getConnection() helps to establish the connection
  - **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/ABC** where **jdbc** is the API, **mysql** is the database, **localhost** is the server name on which mysql is running, we may also use IP address, **3306** is the port number and **ABC**is the database name. We may use any database in MySQL here.
  - **Username:** The default username for the mysql database is **root**.
  - **Password:** It is the password given by the user at the time of installing the mysql database.

WORKING(contd.)

Connection con=**DriverManager**.**getConnection**(
"**jdbc:mysql://localhost:3306/ABC**" , "**root**" , "**root123**");

- The JDBC url used is

jdbc:mysql://localhost:3306/ABC

- Here **jdbc** is the API,
- **mysql** is the database,
- **localhost** is the server name on which **mysql** is running(we can also give IP address here)
- **3306** is the port number
- **ABC** is the database name created in MySQL *(give your database name here)*

- **root** is the username of MySQL(root is the default username . Other MySQL username we can give here).
- **root123** is the password of MySQL given during installation(give password you gave during installation)

WORKING(contd.)

Statement **st**=**con**.createStatement(); // con is the Connection object

- Statement object **st** is created from the Connection object **con** using the method createStatement(). **st** can be used to **send and execute SQL statements** to a database.

- To execute a SQL statement **SELECT** we use **executeQuery()** method on **Statement** object **.**
- (Note:To execute a SQL commands CREATE TABLE, INSERT, DELETE , UPDATE we use executeUpdate() method on **Statement** object**.**)

ResultSet rs=**st**.**executeQuery**("SELECT rollno, name FROM Student");

- SELECT command is executed using **executeQuery()** method on **Statement** object st.
- Result of this SELECT is the rows containing all rollno and name from the table Student in the form of ResultSet.
- **ResultSet** object **rs** maintains a cursor that is initially positioned **before the first row** in the result of SELECT command**.**

WORKING(contd.)

```
ResultSet rs=st.executeQuery("SELECT rollno, name FROM  student");
        while(rs.next())
        {
                System.out.println(rs.getInt(1));
                System.out.println(rs.getString(2));
        }
```

- **ResultSet** object **rs maintains a cursor** that is initially positioned **before the first row in the result.**
- When **while(rs.next()) is first executed** the cursor moves forward to first row of result
  - System.out.println(rs.getInt(1)); prints the value of first attribute (rollno)
  - System.out.println(rs.getString(2)); prints the value of second attribute (name) in the result
- Next time **while** loop is executed, if second row is there, then rs.nex() moves cursor to second row and prints the values in that row.
- This continues until there is no more row in the result.

con.close(); closes the connection

WORKING(contd.)

```
ResultSet rs=st.executeQuery("SELECT rollno, name FROM   student");
        while(rs.next())
        {
                System.out.println(rs.getInt(1));
                System.out.println(rs.getString(2));
        }
```

Working:

- Here rollno and name in all rows in student table are retrieved.
- **ResultSet** object **rs** maintains a cursor that is initially positioned **before the first row in the result.**
- When **while(rs.next()) is first executed** the cursor moves forward to first row of result and prints the value of first attribute (rollno) and second attribute(name)in first row in the result
- Next time while loop is executed, if second row is there, then rs.nex() moves cursor to second row and prints the values in that row.
- This continues until there is no more row in the result.

con.close(); closes the connection

WORKING(contd.)

- ClassNotFoundException , SQLException and other exceptions may occur during these steps.
- ClassNotFoundException is thrown from main function

```
public static void main(String args[]) throws
    ClassNotFoundException
```

- Using try catch other exceptions like SQLException can be handled

```
try{

   }catch(SQLException se)
   {    }
catch(Exception e)
   {    }
```

SELECT rows based on conditionusing Java

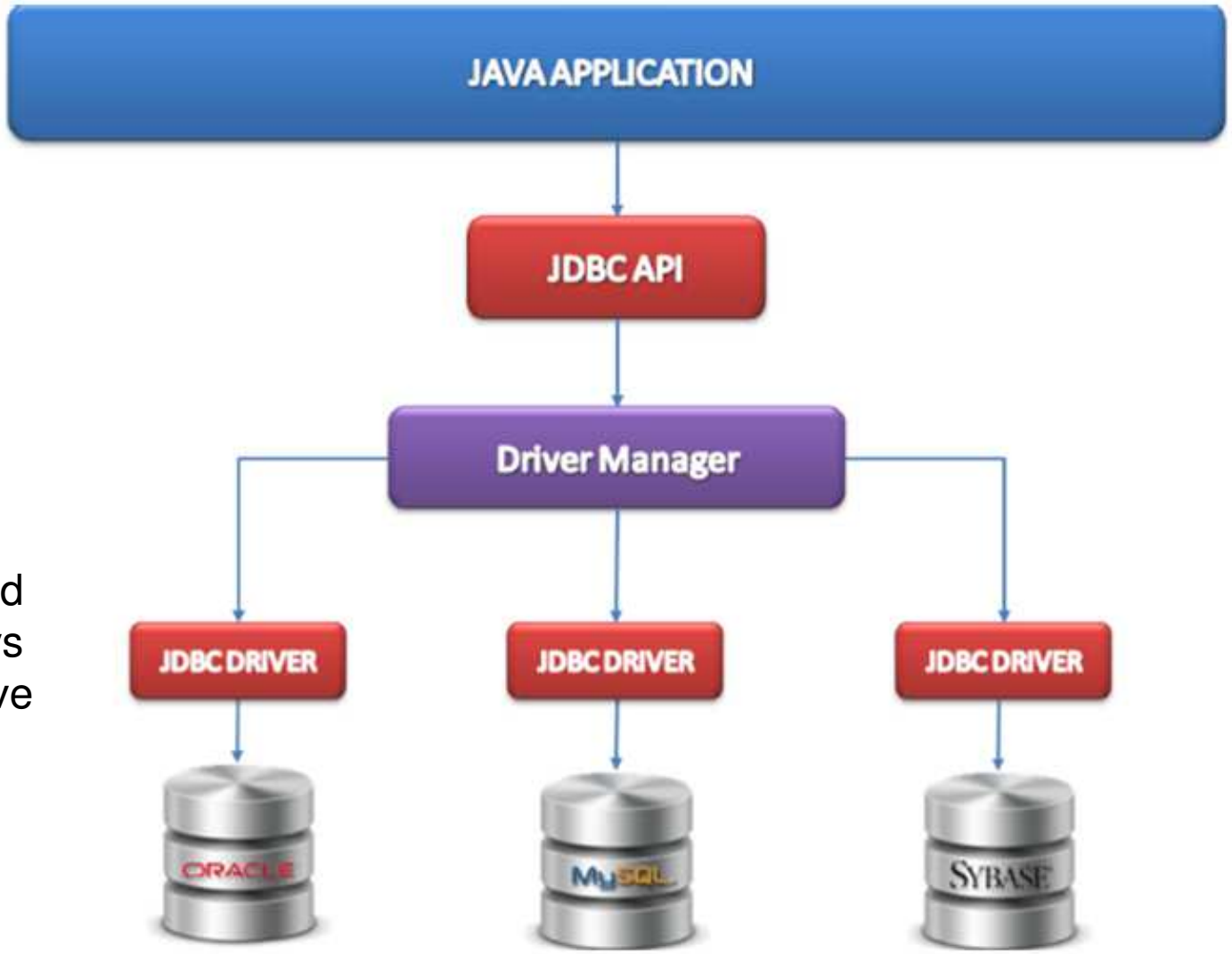
- Write a Java program to **list name of the student** in **Student** table **having given roll number**

SELECT row based on condition Using Java

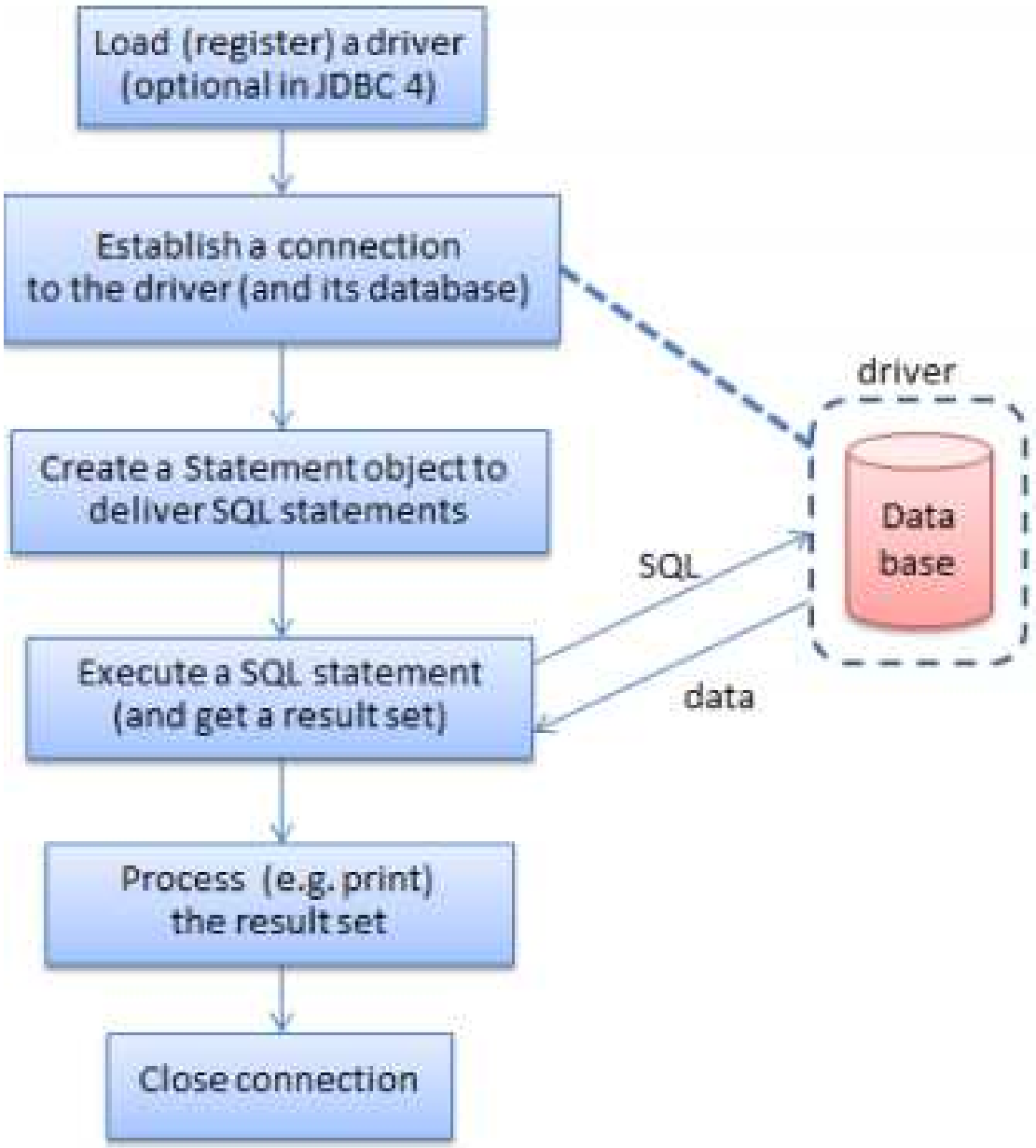
```
import java.sql.*;
import java.util.*
public class InsertEg
{   public static void main(String args[]) throws ClassNotFoundException
    { try { int roll;
            Scanner sc=new Scanner(System.in);
            Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/ABC" , "root","root123");
            Statement st=con.createStatement();
            System.out.println("Enter the rollno of student ");
            roll=sc.nextInt();
ResultSet  rs=st.executeQuery("SELECT  name  FROM  student  WHERE
    rollno='"+ roll +"' ");
            System.out.println("Name of student is “+rs.getString(1));

            con.close();
              }catch(SQLException e) {      System.out.println("Error is " +e);    }
    }
}
```

Summary

- Programming Language -**Java**
  - for coding, developing interfaces(**front end** of an application)
- Database – **MySQL ,** Oracle , PostgreSQL
  - For storing data- (**back end** of an application)
  - Relational database -Tabular structure
    - **E.g.** - Oracle, Microsoft Access, **MySQL**, PostgreSQL, MongoDB etc
- SQL- Structured Query Language
  - **SQL** statements are used to perform operations on a database. CREATE TABLE, INSERT, DELETE, UPDATE, SELECT



Com.jdbc.mysql.Driver


Load (register) a driver
(optional in JDBC 4)
Establish a connection
to the driver (and its database)
Create a Statement object to
deliver SQL statements
Execute a SQL statement
(and get a result set)
Process (e.g. print)
the result set
Close connection
driver
Data
base
SQL
data

Thank You